

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

А. В. Нікітчук

ПРОГРАМОВАНІ ЗАСОБИ В ІНТЕЛЕКТУАЛЬНІЙ РАДІОЕЛЕКТРОННІЙ ТЕХНІЦІ

Комп'ютерний практикум

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньою програмою «Інтелектуальні технології радіоелектронної техніки»
спеціальності G5 «Електроніка, електронні комунікації,
приладобудування та радіотехніка»

Електронне мережеве навчальне видання

Київ
КПІ ім. ІГОРЯ СІКОРСЬКОГО
2026

УДК 004.4'23.031.6:004.89(075.8)
Н 62

Автор *Нікітчук Артем Валерійович*

Рецензент *Мирончук О. Ю., PhD, доцент кафедри РТС РТФ*

Відповідальний редактор *Шульга А. В., к. т. н., доцент*

*Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 8 від 04.06.2026 р.)
за поданням вченої ради радіотехнічного факультету
(протокол № 08/2026 від 26.05.2026 р.)*

Нікітчук А. В.

Н 62 Програмовані засоби в інтелектуальній радіоелектронній техніці. Комп'ютерний практикум [Електронний ресурс] : навч. посіб. для здобувачів ступеня бакалавра за освіт. програмою «Інтелектуальні технології радіоелектронної техніки» спец. G5 «Електроніка, електронні комунікації, приладобудування та радіотехніка» / А. В. Нікітчук ; КПІ ім. Ігоря Сікорського. – Електрон. текст. дані (1 файл). – Київ : КПІ ім. Ігоря Сікорського, 2026. – 202 с.

Посібник містить набір практичних робіт, присвячених програмуванню засобів інтелектуальної радіоелектронної техніки та охоплює широкий спектр тем, від опанування середовища розробки та конструкцій мови програмування до використання мережових інтерфейсів, впровадження операційних систем реального часу та розгортання моделей машинного навчання. Викладений матеріал спрямований на розвиток практичних навичок розробки й оптимізації програмного забезпечення для програмованих засобів з обмеженими обчислювальними ресурсами. Завдання структуровані за принципом зростання складності. Кожна робота містить покроковий алгоритм виконання та посилання на актуальну технічну документацію. Видання призначене для здобувачів вищої освіти за освітньою програмою «Інтелектуальні технології радіоелектронної техніки» спеціальності G5 «Електроніка, електронні комунікації, приладобудування та радіотехніка».

УДК 004.4'23.031.6:004.89(075.8)

Реєстр. № НП 25/26-414. Обсяг 11,44 авт. арк.
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Берестейський, 37, м. Київ, 03056
<https://kpi.ua>

Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

© А. В. Нікітчук, 2026
© КПІ ім. Ігоря Сікорського, 2026

ЗМІСТ

Вступ	4
Практична робота № 1. Середовище розробки та симуляції	5
Практична робота № 2. Оператори та підпрограми	27
Практична робота № 3. Контроль потоку програми	37
Практична робота № 4. Цикли, часові затримки та таймери	48
Практична робота № 5. Комунікація з периферійними пристроями за протоколами UART, SPI та I2C	61
Практична робота № 6. Підключення до бездротової мережі та визначення точного мережевого часу.....	87
Практична робота № 7. Клієнт-серверна комунікація за протоколом HTTP	98
Практична робота № 8. Застосування операційної системи реального часу (RTOS).....	111
Практична робота № 9. Синхронізація доступу до спільних ресурсів в RTOS	127
Практична робота № 10. Оптимізація програми за допомогою прямого звернення до регістрів мікроконтролера	138
Практична робота № 11. Підвищення ефективності обчислень та зменшення часу виконання.....	145
Практична робота № 12. Забезпечення енергоефективності та використання режимів зниженого енергоспоживання	160
Практична робота № 13. Машинне навчання моделі для класифікації динамічних процесів.....	172
Практична робота № 14. Розгортання моделі штучного інтелекту на мікроконтролері	190
Перелік посилань	201

ВСТУП

Інтелектуальна радіoeлектронна техніка є основою промислової автоматизації, телекомунікацій, робототехніки та Інтернету речей. Впровадження інтелектуальних функцій потребує складних програмно-апаратних комплексів, розробка яких вимагає опанування сучасних програмованих засобів, технологій створення вбудованого програмного забезпечення та розуміння принципів його функціонування.

Основною метою посібника є формування практичних навичок розробки та оптимізації програмного забезпечення для мікроконтролерних систем, а також ознайомлення з сучасними підходами до створення інтелектуальних вбудованих пристроїв.

У процесі виконання практичних робіт здобувачі освіти ознайомляться із середовищем розробки та симуляції, конструкціями мови програмування, принципами керування потоком виконання програм, використанням таймерів і часових затримок, а також із засобами взаємодії мікроконтролера з периферійними пристроями та бездротовими мережами через поширені протоколи обміну даними. Методична основа частини робіт базується на адаптованих матеріалах курсу «Програмування вбудованих систем Інтернету речей» [1, 2].

Також увага приділяється застосуванню операційних систем реального часу та оптимізації програмного коду для підвищення продуктивності та енергоефективності.

Завершальні роботи охоплюють основи машинного навчання та розгортання моделей штучного інтелекту на пристроях з обмеженими ресурсами, що дозволяє сформулювати уявлення про сучасні напрями розвитку інтелектуальних радіoeлектронних систем.

Виконання запропонованих завдань сприяє розвитку системного інженерного мислення та формуванню фахових компетенцій, необхідних для розробки високотехнологічних рішень у галузі інтелектуальної радіoeлектронної техніки.

ПРАКТИЧНА РОБОТА № 1

СЕРЕДОВИЩЕ РОЗРОБКИ ТА СИМУЛЯЦІЇ

Мета: Набути навичок налаштування інтегрованого середовища розробки для створення та налагодження програмного забезпечення.

План проведення практичного заняття

- Опрацювання теоретичних матеріалів.
 - Покрокове виконання завдань.
 - Подання результатів (звіту) на перевірку.
-

Теоретичні відомості

Сучасні підходи до проектування інтелектуальної техніки передбачають широке використання віртуального прототипування, що дозволяє здійснювати ранню верифікацію програмних рішень, ще до інтеграції коду в цільову апаратну платформу, та забезпечує ефективну взаємодію в межах розподілених команд.

Для цього використовують спеціалізовані симулятори, що дозволяють виконувати код у віртуальному середовищі без потреби у фізичному обладнанні. Інструменти на кшталт QEMU чи Proteus орієнтовані на глибокий технічний аналіз, профілювання ресурсів та складну емуляцію схемотехніки. Натомість онлайн-платформи забезпечують візуальну перевірку логіки роботи пристрою та взаємодії програмного коду з периферією.

Особливої уваги заслуговує Wokwi – онлайн-платформа для симуляції, що підтримує архітектури RISC-V, Xtensa, ARM та AVR та велику кількість датчиків, індикаторів та інших периферійних пристроїв [3]. Її головною перевагою інструменту є робота безпосередньо у веб-браузері, що дозволяє виконувати розробку та тестування без потреби в локальному інсталюванні компіляторів чи середовищ розробки. Wokwi забезпечує сумісність з екосистемою Arduino, зокрема підтримуючи файли формату «.ino». Використання цієї платформи дозволяє проводити безпечно відлагодження коду без ризику пошкодження фізичного

обладнання, а можливість надання доступу до проєкту через URL-посилання оптимізує процеси командної розробки та технічного рецензування.

Незважаючи на високу ефективність, можливостей онлайн-середовища може бути недостатньо коли завдання ускладнюється і час збірки проєкту на віддаленому сервері зростає, а також, коли необхідно працювати з великою кількістю файлів, їх версіями, великими бібліотеками тощо. З огляду на ці фактори, для розробки високоефективних програмованих засобів доцільно переходити до використання локальних професійних інтегрованих середовищ розробки, таких як Visual Studio Code (VS Code).

VS Code – це сучасне, кросплатформове середовище, яке завдяки розширенням PlatformIO та Wokwi набуває функціональності для роботи з програмованими засобами та вбудованими системами [4].

PlatformIO – це професійна, кросплатформна система автоматизації збірки, що створена з метою стандартизувати процес розробки для максимально широкого спектру мікроконтролерів та вбудованих систем. PlatformIO дозволяє швидко скопіювати код на власному ПК. Розширення Wokwi використовує згенерований файл прошивки та дозволяє запустити симуляцію у вікні VS Code. Таке поєднання усуває залежність від завантаження віддалених серверів.

Завдання

1. Зареєструватись на платформі Wokwi. Ознайомитись з технічною документацією сервісу.
 2. Створити проєкт (на базі Arduino Uno). Зібрати схему та розробити програму, що реалізує блимання групою світлодіодів відповідно до індивідуального варіанта (табл. 1.1).
 3. Налаштувати середовище VS Code для локальної збірки та запуску створеної програми (використовуючи фреймворк PlatformIO та розширення Wokwi).
-

Таблиця варіантів

Таблиця 1.1. Перелік груп світлодіодів за варіантами

Варіант №	Світлодіоди
1	червоний, помаранчевий, жовтий, зелений, синій, рожевий
2	червоний, зелений, синій, рожевий
3	жовтий, зелений, рожевий
4	червоний, жовтий, синій
5	червоний, помаранчевий, жовтий
6	синій, рожевий
7	помаранчевий, жовтий, зелений, синій, рожевий
8	червоний, жовтий, синій, рожевий
9	жовтий, зелений, синій
10	помаранчевий, зелений, рожевий
11	зелений, рожевий
12	червоний, жовтий, зелений, синій, рожевий
13	червоний, жовтий, зелений, рожевий
14	помаранчевий, синій, рожевий
15	червоний, жовтий, зелений
16	жовтий, рожевий
17	жовтий, синій
18	червоний, помаранчевий, зелений, синій, рожевий
19	червоний, жовтий, зелений, синій
20	червоний, помаранчевий, зелений, рожевий
21	помаранчевий, жовтий, рожевий
22	зелений, синій
23	червоний, помаранчевий, жовтий, синій, рожевий
24	червоний, помаранчевий, синій, рожевий
25	помаранчевий, зелений, синій
26	жовтий, зелений
27	помаранчевий, рожевий
28	червоний, помаранчевий, жовтий, зелений, рожевий
29	червоний, помаранчевий, жовтий, рожевий
30	помаранчевий, жовтий, зелений

Продовження таблиці 1.1

Варіант №	Світлодіоди
31	помаранчевий, синій
32	червоний, помаранчевий, жовтий, зелений, синій
33	червоний, помаранчевий, зелений, синій
34	помаранчевий, жовтий, синій
35	жовтий, зелений, синій, рожевий
36	червоний, помаранчевий, рожевий
37	помаранчевий, зелений
38	помаранчевий, жовтий
39	помаранчевий, жовтий, синій, рожевий
40	червоний, помаранчевий, жовтий, зелений
41	червоний, зелений, рожевий
42	червоний, рожевий
43	помаранчевий, зелений, синій, рожевий
44	червоний, помаранчевий, жовтий, синій
45	червоний, синій, рожевий
46	червоний, помаранчевий, синій
47	червоний, синій
48	червоний, зелений
49	червоний, жовтий
50	помаранчевий, жовтий, зелений, рожевий
51	зелений, синій, рожевий
52	червоний, зелений, синій
53	червоний, помаранчевий
54	помаранчевий, жовтий, зелений, синій
55	жовтий, синій, рожевий
56	червоний, жовтий, рожевий
57	червоний, помаранчевий, зелений
58	червоний, помаранчевий, жовтий, зелений, синій, рожевий
59	червоний, зелений, синій, рожевий
60	жовтий, зелений, рожевий

Порядок виконання роботи

Крок 1. Реєстрація та підготовка до роботи:

- а. Перейдіть на сайт Wokwi (URL: <https://wokwi.com/>) та зареєструйте власний обліковий запис одним із наведених способів (рис. 1.1).

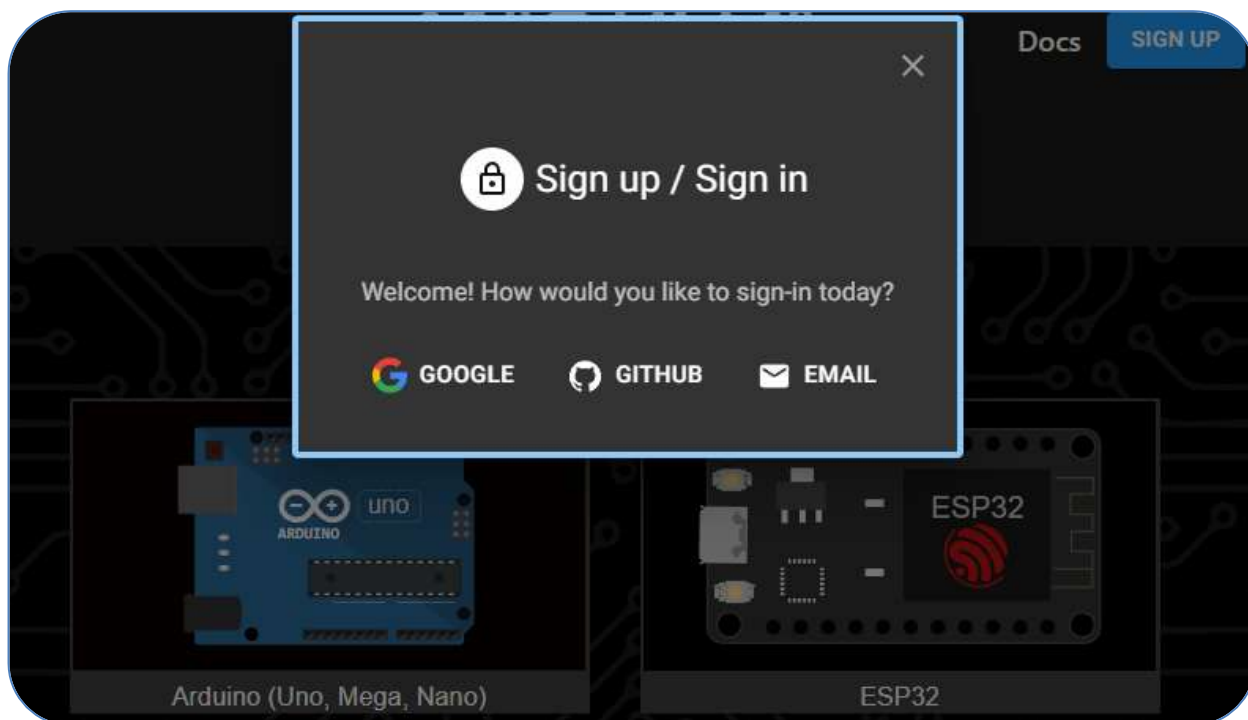


Рис. 1.1. Вікно входу/реєстрації користувача

- б. Ознайомтесь з основними розділами документації Wokwi за посиланням (URL: <https://docs.wokwi.com/>):
- підтримуване обладнання;
 - редактор діаграм;
 - робота з редактором коду;
 - підключення бібліотек.

Крок 2. Ініціалізація проекту в середовищі симуляції:

- а. Після авторизації відкрийте головне меню та перейдіть до розділу «My Projects» (рис. 1.2).

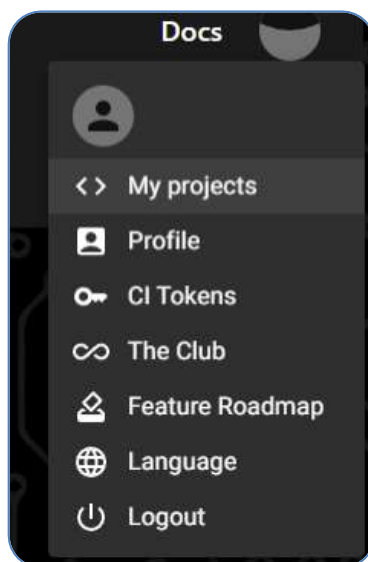


Рис. 1.2. Меню середовища симуляції

- б. Створіть новий проєкт, натиснувши кнопку «**NEW PROJECT**» зображену на рисунку 1.3.

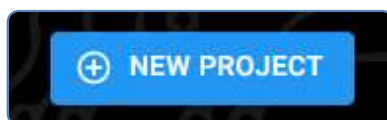


Рис. 1.3. Кнопка «NEW PROJECT»

Примітка. При використанні безкоштовного тарифного плану всі збережені проєкти стають публічними. Збережені роботи інших користувачів можливо переглянути в розділах присвячених кожній з підтримуваних платформ.

- с. У переліку доступних платформ оберіть «**Arduino Uno**» (рис. 1.4).



Рис. 1.4. Модель плати мікроконтролера Arduino Uno

Крок 3. Формування віртуальної апаратної частини:

- а. У вікні симуляції (Simulation) натисніть кнопку «+» для додавання компонентів (рис. 1.5).



Рис. 1.5. Елементи управління симуляцією

- б. У меню оберіть світлодіод «LED» із переліку базових компонентів (рис. 1.6).

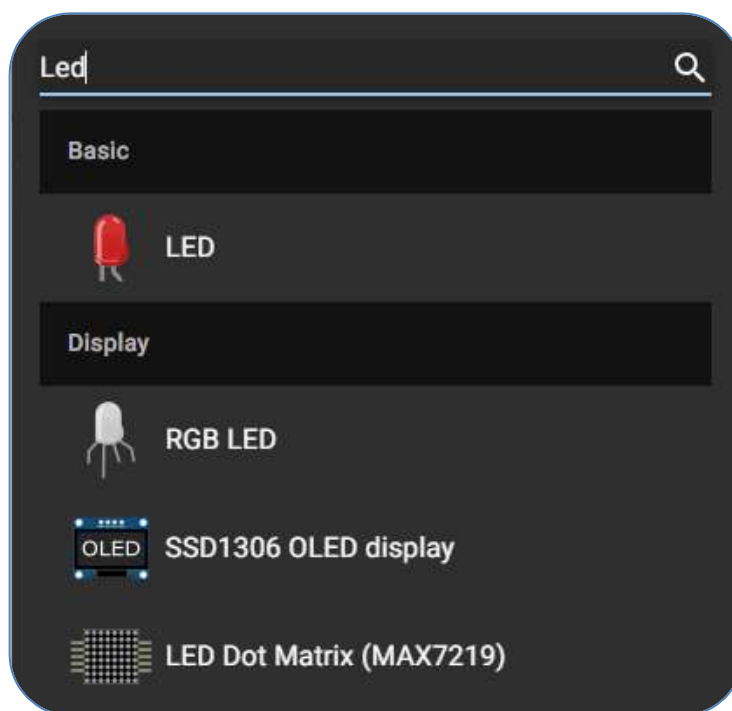


Рис. 1.6. Список доступних електронних компонентів

Для зміни параметрів (кольору або орієнтації) використовуйте меню властивостей виділеного компонента.

- с. Побудуйте схему згідно з рис. 1.7 (у середовищі симуляції допускається нехтування обмежувальними резисторами для спрощення схеми).

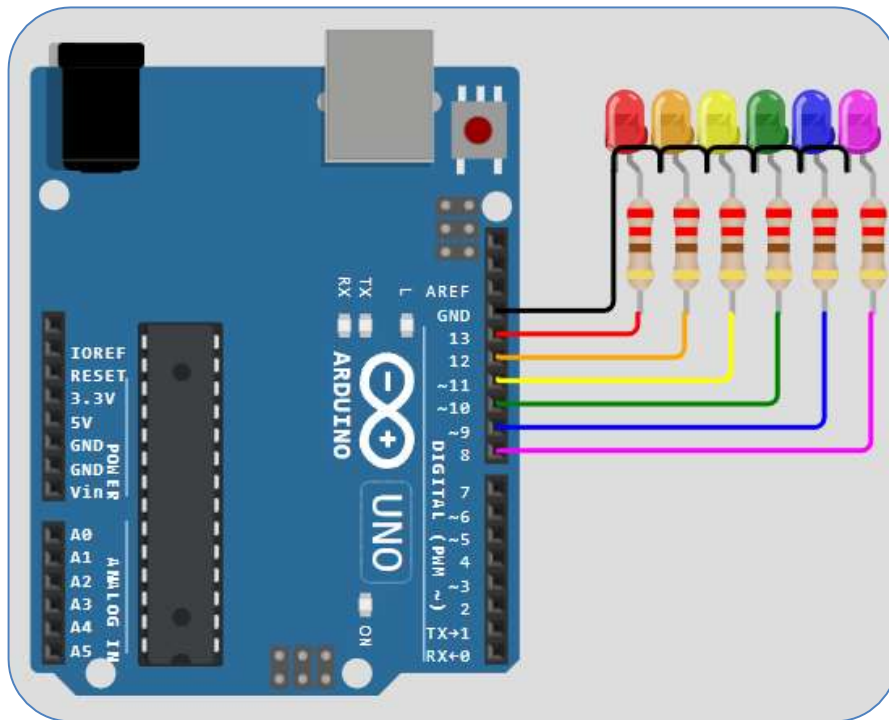


Рис. 1.7. Вигляд зібраної схеми

Для автоматизації конфігурування апаратної частини можна скористатися файлом «**diagram.json**», де зберігаються параметри всіх з'єднань у текстовому форматі. Для отримання ідентичної схеми (рис. 1.7) замініть вміст вкладки «**diagram.json**» наведеним нижче текстом:

```
{
  "version": 1,
  "editor": "wokwi",
  "parts": [
    {
      "type": "wokwi-arduino-uno",
      "id": "arduino",
      "top": 36.9,
      "left": -179.5,
      "rotate": 90,
      "attrs": {}
    },
    {
      "type": "wokwi-led",
      "id": "diode1",
```

```
    "top": 17.1,
    "left": 82.1,
    "attrs": { "color": "red" }
  },
  {
    "type": "wokwi-led",
    "id": "diode2",
    "top": 17.1,
    "left": 101.3,
    "attrs": { "color": "orange" }
  },
  {
    "type": "wokwi-led",
    "id": "diode3",
    "top": 17.1,
    "left": 120.5,
    "attrs": { "color": "yellow" }
  },
  {
    "type": "wokwi-led",
    "id": "diode4",
    "top": 17.1,
    "left": 139.7,
    "attrs": { "color": "green" }
  },
  {
    "type": "wokwi-led",
    "id": "diode5",
    "top": 17.1,
    "left": 158.9,
    "attrs": { "color": "blue" }
  },
  {
    "type": "wokwi-led",
```

```
    "id": "diode6",
    "top": 17.1,
    "left": 178.1,
    "attrs": { "color": "magenta" }
  },
  {
    "type": "wokwi-resistor",
    "id": "resistor1",
    "top": 81.6,
    "left": 76.25,
    "rotate": 90,
    "attrs": { "value": "220" }
  },
  {
    "type": "wokwi-resistor",
    "id": "resistor2",
    "top": 81.6,
    "left": 95.45,
    "rotate": 90,
    "attrs": { "value": "220" }
  },
  {
    "type": "wokwi-resistor",
    "id": "resistor3",
    "top": 81.6,
    "left": 114.65,
    "rotate": 90,
    "attrs": { "value": "220" }
  },
  {
    "type": "wokwi-resistor",
    "id": "resistor4",
    "top": 81.6,
    "left": 133.85,
```

```

    "rotate": 90,
    "attrs": { "value": "220" }
  },
  {
    "type": "wokwi-resistor",
    "id": "resistor5",
    "top": 81.6,
    "left": 153.05,
    "rotate": 90,
    "attrs": { "value": "220" }
  },
  {
    "type": "wokwi-resistor",
    "id": "resistor6",
    "top": 81.6,
    "left": 172.25,
    "rotate": 90,
    "attrs": { "value": "220" }
  }
],
"connections": [
  [ "diode1:C", "arduino:GND.1", "black", [ "v57.6", "h-47.6" ] ],
  [ "diode6:C", "diode5:C", "black", [ "v-9.6", "h-18.8" ] ],
  [ "diode5:C", "diode4:C", "black", [ "v-9.6", "h-18.8" ] ],
  [ "diode4:C", "diode3:C", "black", [ "v-9.6", "h-18.8" ] ],
  [ "diode3:C", "diode2:C", "black", [ "v-9.6", "h-18.8" ] ],
  [ "diode2:C", "diode1:C", "black", [ "v-9.6", "h-18.8" ] ],
  [ "diode1:A", "resistor1:1", "red", [ "v0" ] ],
  [ "resistor1:2", "arduino:13", "red", [ "v0", "v8.4", "h-57.6" ] ],
  [ "diode2:A", "resistor2:1", "orange", [ "v0" ] ],
  [ "resistor2:2", "arduino:12", "orange", [ "v18", "h-76.8" ] ],
  [ "diode3:A", "resistor3:1", "yellow", [ "v0" ] ],
  [ "resistor3:2", "arduino:11", "yellow", [ "v27.6", "h-96" ] ],
  [ "diode4:A", "resistor4:1", "green", [ "v0" ] ],

```

```
[ "resistor4:2", "arduino:10", "green", [ "v37.2", "h-115.2" ] ],
[ "diode5:A", "resistor5:1", "blue", [ "v0" ] ],
[ "resistor5:2", "arduino:9", "blue", [ "v46.8", "h-134.4" ] ],
[ "diode6:A", "resistor6:1", "magenta", [ "v0" ] ],
[ "resistor6:2", "arduino:8", "magenta", [ "v56.4", "h-153.6" ] ]
]
}
```

Крок 4. Програмування та верифікація алгоритму:

- а. Додайте у редактор Wokwi наведений код, що реалізує циклічне перемикавання логічних рівнів на виході контролера:

```
void setup()
{
    // Конфігурація цифрового виводу №12 як виходу (OUTPUT)
    pinMode(12, OUTPUT);
}

void loop()
{
    // Встановлення високого рівня напруги на виводі №12,
    // у випадку Arduino Uno – це +5 В
    // Через світлодіод починає протікати електричний струм
    digitalWrite(12, HIGH);

    delay(500); // Програмна затримка виконання на 500 мс (0.5
секунд)

    // Встановлення низького рівня напруги (0 В) на виводі №12
    // Світлодіод вимикається
    digitalWrite(12, LOW);

    delay(500); // Програмна затримка виконання на 0.5 с
}
```

- b. Запустіть процес симуляції за допомогою кнопки зі стрілкою (рис. 1.5) та переконайтеся у працездатності програми (помаранчевий світлодіод повинен блимати).
- c. Модифікуйте вихідний код згідно з вимогами індивідуального варіанта завдання № 2.
- d. Збережіть фінальну версію проєкту та скопіюйте URL-посилання для звітності.

Крок 5. Налаштування середовища для локальної збірки та симуляції:

- a. Встановіть середовище VS Code. Для завантаження перейдіть на офіційний сайт <https://code.visualstudio.com/>.
- b. Відкрийте VS Code. Натисніть на іконку Extensions (квадратики) на лівій бічній панелі (рис. 1.8) або Ctrl+Shift+X.

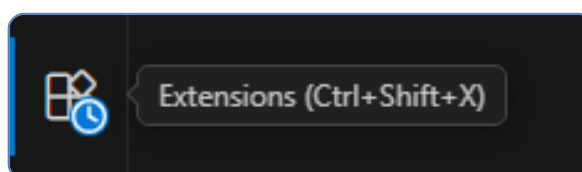


Рис. 1.8. Менеджер розширень

- c. У рядку пошуку введіть «**PlatformIO IDE**», знайдіть його в списку та натисніть кнопку **Install** (рис. 1.9).

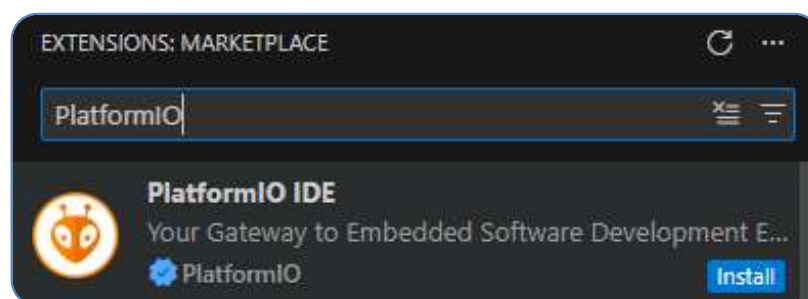


Рис. 1.9. Розширення PlatformIO

- d. У рядку пошуку введіть «**Wokwi**», оберіть перший результат і натисніть **Install** (рис. 1.10).

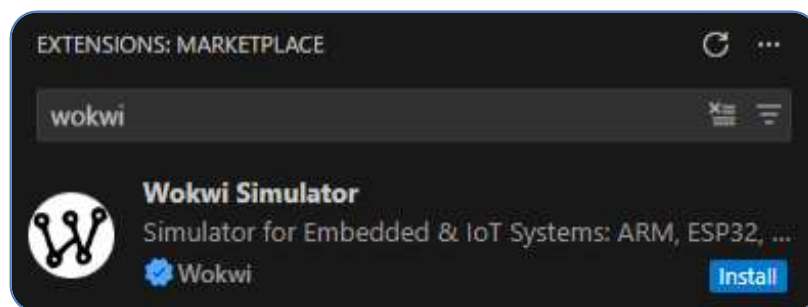


Рис. 1.10. Розширення Wokwi

- е. Для створення нового проекту, після встановлення розширень, на бічній панелі з'явиться іконка мурахи (PlatformIO), натисніть на неї та оберіть «**New Project**» (рис. 1.11).

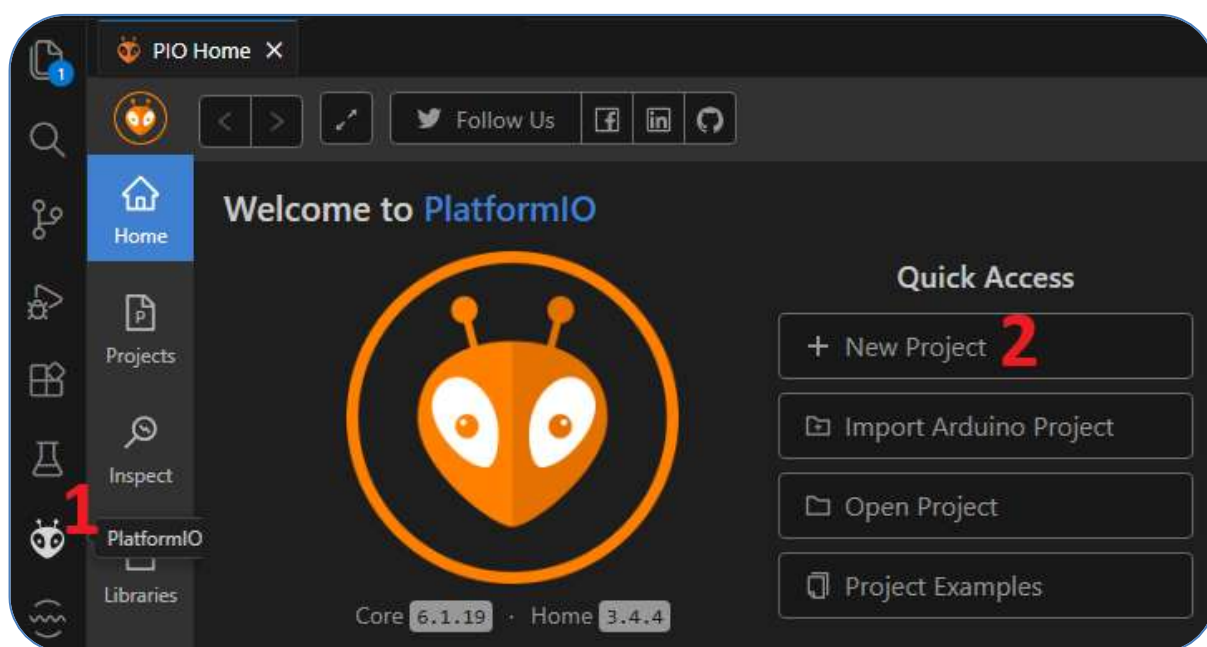


Рис. 1.11. Домашня сторінка PlatformIO (PIO Home)

У вікні налаштувань (рис. 1.12) введіть назву (Name) проекту у поле, – оберіть модель плати (Board) **Arduino Uno**, оберіть фреймворк (Framework) **Arduino** та локацію (Location) для збереження проекту.

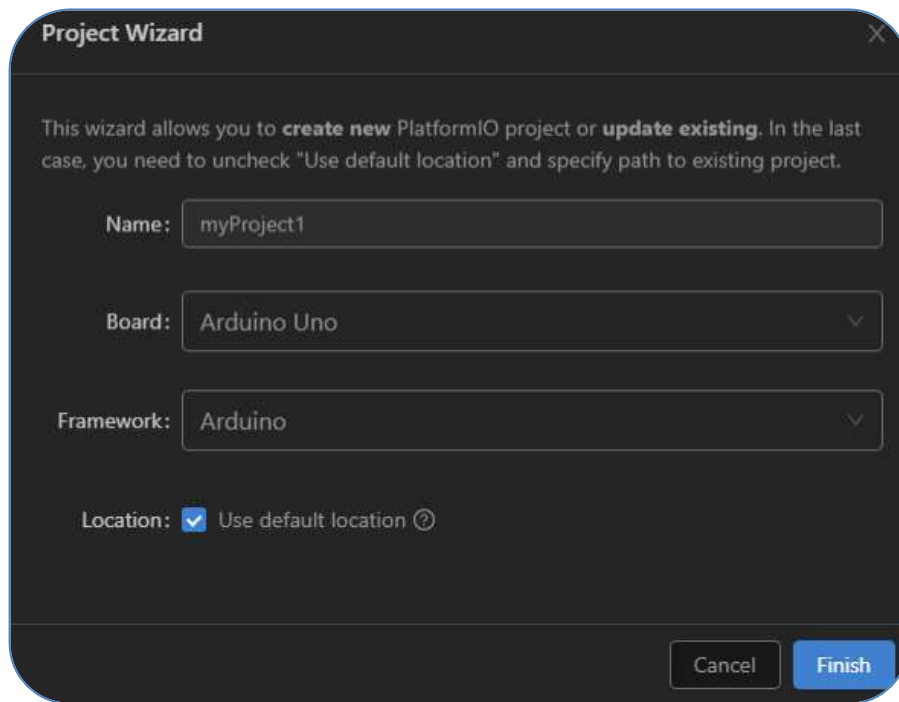


Рис. 1.12. Налаштування нового проекту

Натисніть Finish і зачекайте, поки проект завантажиться.

- f. У провіднику проектів знайдіть та відкрийте файл `src/main.cpp` (рис. 1.13). Вставте свій код у цей файл. Обов'язково додайте на початку рядок `#include <Arduino.h>`.

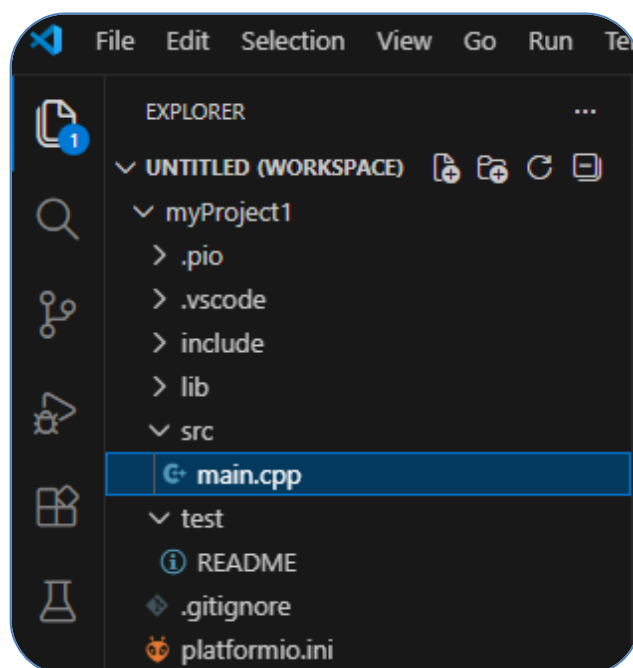


Рис. 1.13. Вміст проекту

- g. Натисніть іконку **Checkmark** (галочка), щоб зібрати проект (Build) (рис. 1.14).

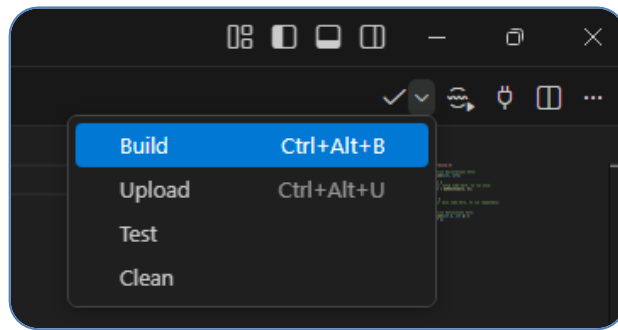


Рис. 1.14. Збірка проекту

- h. Щоб симулятор «знав», що і як запускати, потрібно додати (рис. 1.15) у проект два файли **wokwi.toml** та **diagram.json**.

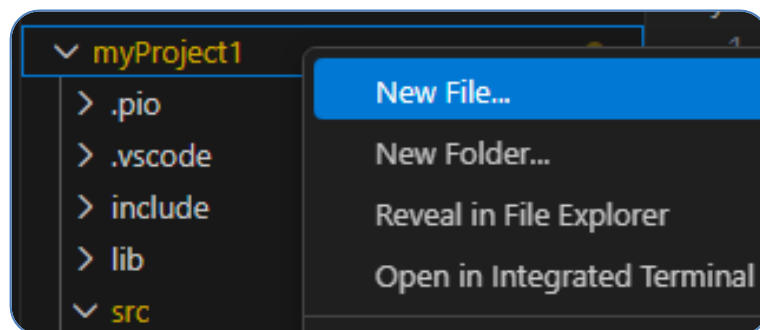


Рис. 1.15. Додавання файлу до проекту

wokwi.toml – вказує шлях до бінарного файлу проекту. Створіть новий файл із такою назвою та додайте до нього шлях до скомпільованих бінарних файлів (.bin та .elf) (рис. 1.16), зазвичай вони знаходяться у теці збірки PlatformIO (рис. 1.17).

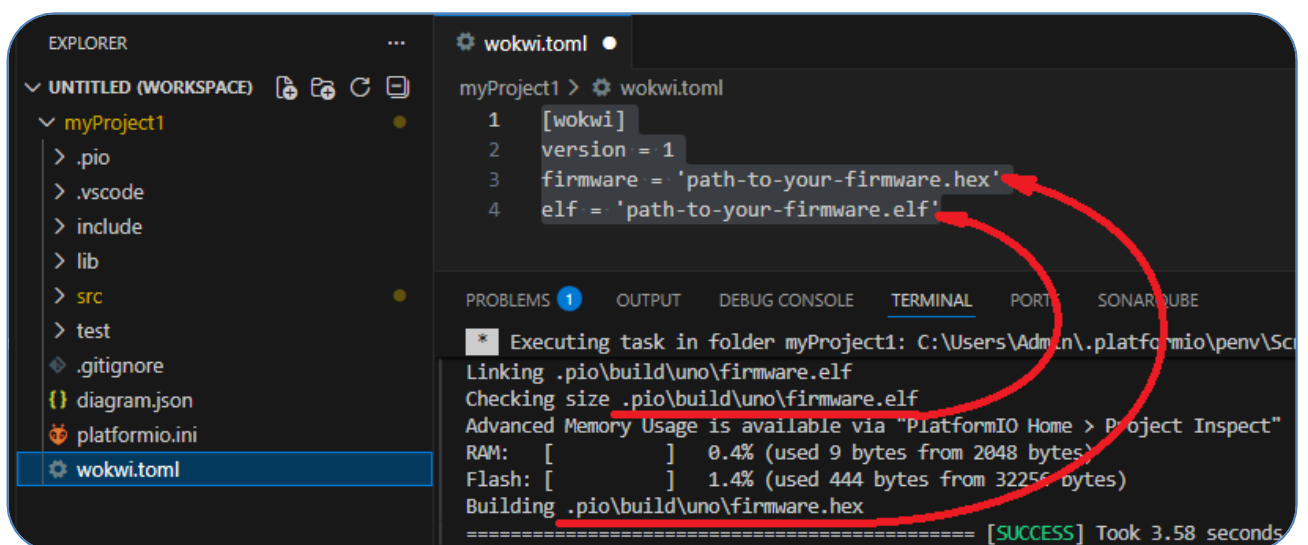
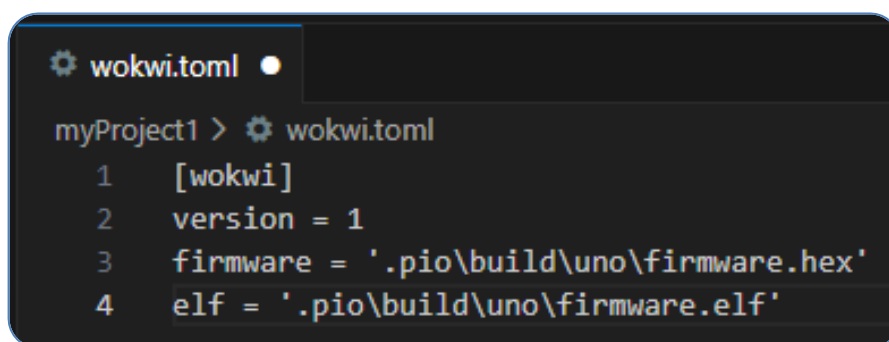


Рис. 1.16. Створення файлу wokwi.toml



```

wokwi.toml
myProject1 > wokwi.toml
1  [wokwi]
2  version = 1
3  firmware = '.pio\build\uno\firmware.hex'
4  elf = '.pio\build\uno\firmware.elf'

```

Рис. 1.17. Вміст файлу wokwi.toml для проекту Arduino Uno

diagram.json – описує апаратну частину (компоненти та з'єднання). При використанні безкоштовного плану Wokwi Community – редагування діаграми у VS Code недоступне, тому рекомендується зібрати схему в онлайн-редакторі Wokwi та скопіювати зміст **diagram.json** звідти (рис. 1.18).

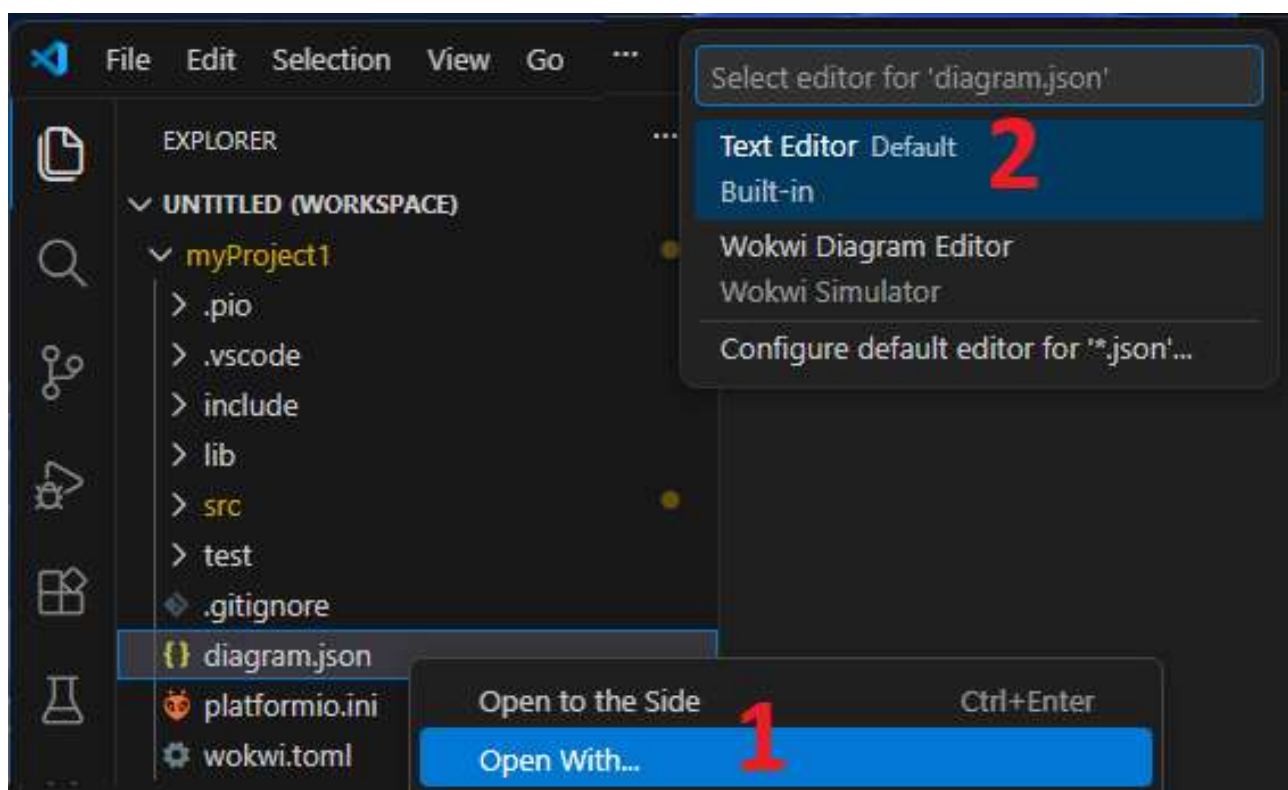


Рис. 1.18. Редагування diagram.json

Після збереження, відкрийте файл **diagram.json** двічі клікнувши по ньому. При першій спробі запуску з'явиться повідомлення про відсутність ліцензії. Натисніть «**Get your license key**». Після перенаправлення у браузер натисніть «**Get your license**» (рис. 1.19), а потім дозвольте відкрити VS

Code. Після активації (Community mode) рекомендується перезапустити VS Code.

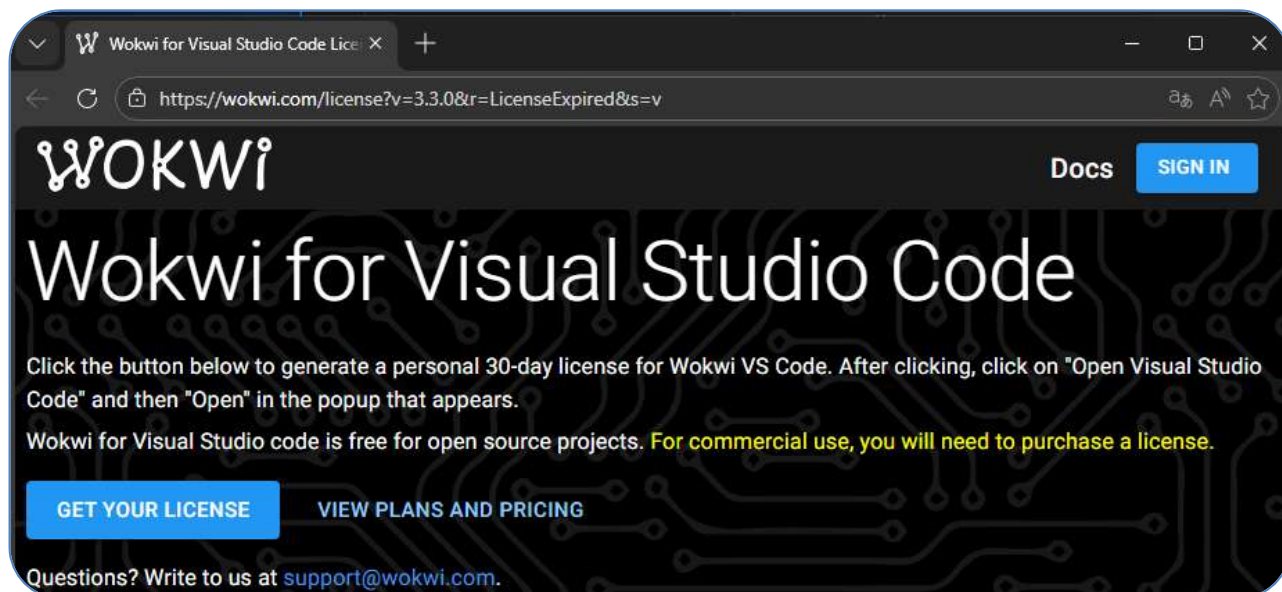


Рис. 1.19. Активація ліцензії Wokwi

Після активації ліцензії вікно Wokwi буде доступне прямо у VS Code (рис. 1.20).

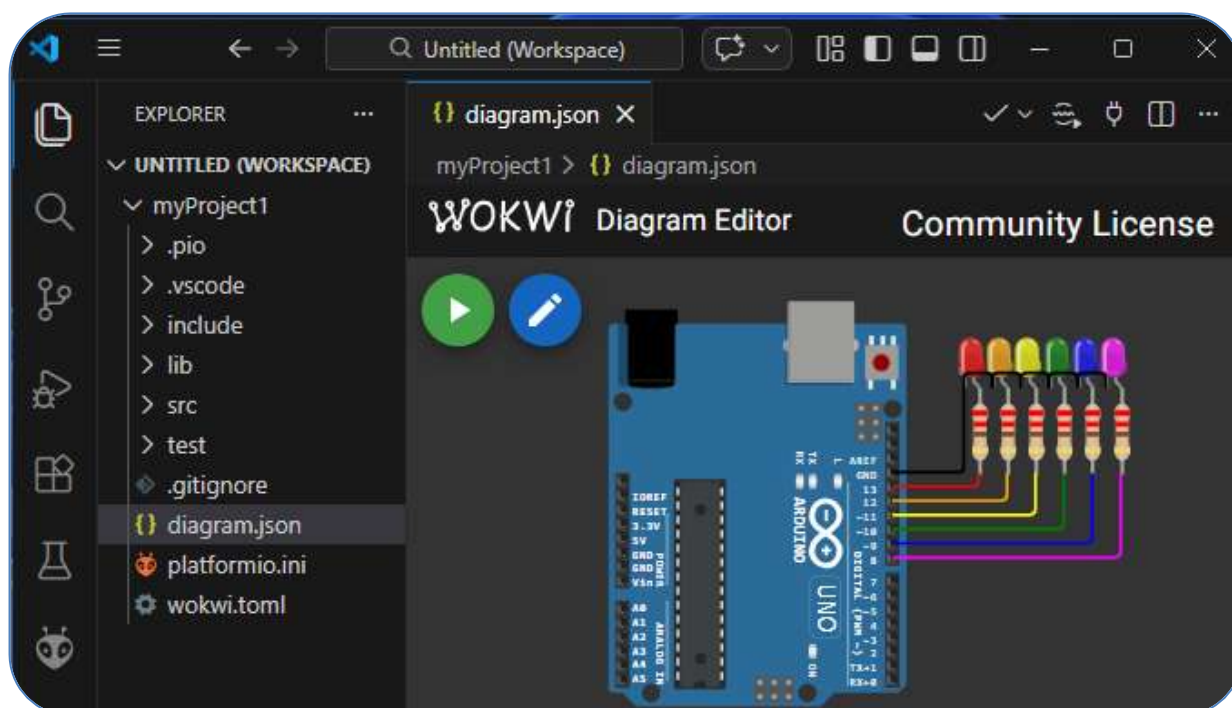


Рис. 1.20. Wokwi у VS Code

- i. Запустіть симуляцію натиснувши на зелену кнопку зі стрілочкою.

Важливо! Для підключення бібліотек (наприклад, для LCD-дисплея) – необхідно перейти у вкладку **Libraries** в меню **PlatformIO**, знайдіть потрібну бібліотеку (наприклад, **LiquidCrystal**) і натисніть «Add to project» (рис. 1.21-1.22).

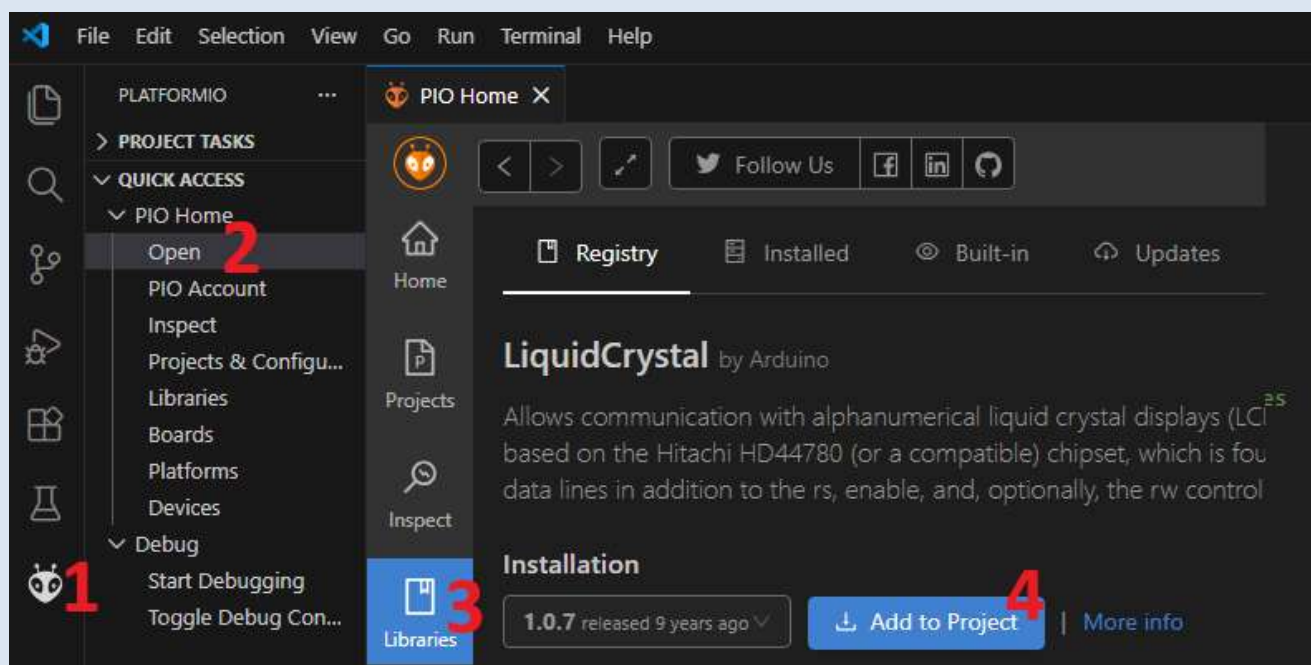


Рис. 1.21. Додавання бібліотек до проекту

Або вручну додайте назву бібліотеки у файл **platformio.ini** під параметром **lib_deps**.

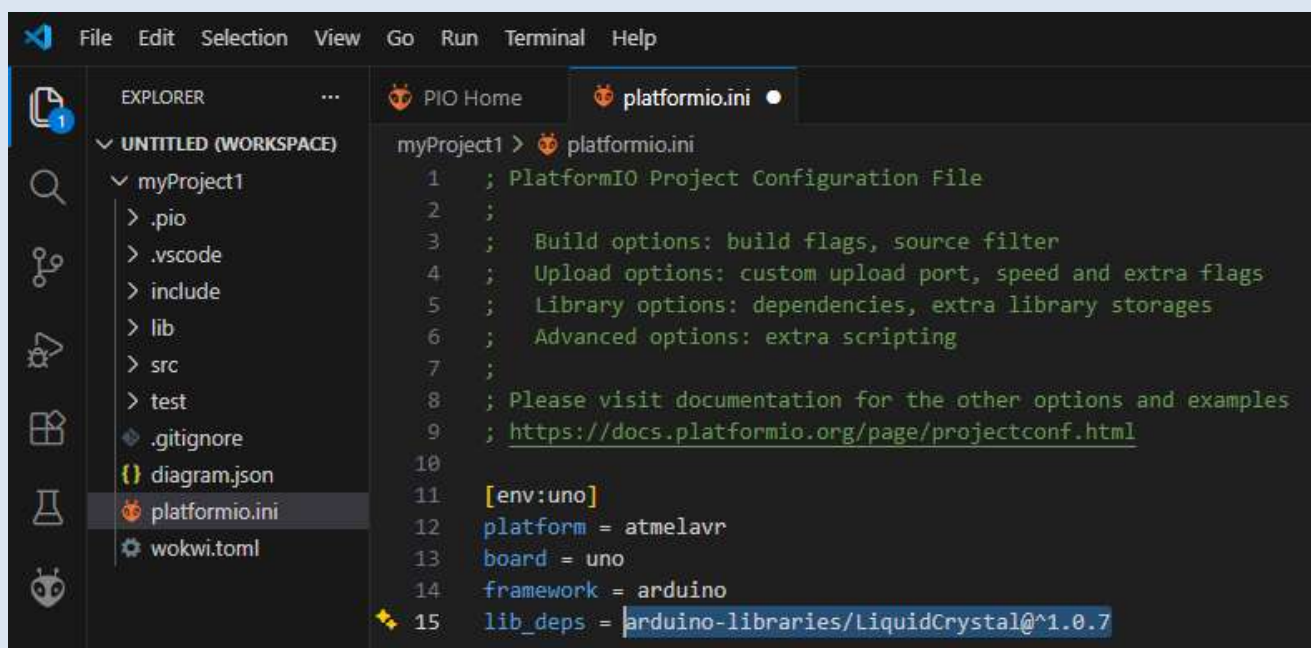


Рис. 1.22. Налаштування бібліотек у проекті

Крок 6. *(Не обов'язковий)* Міграція на альтернативні апаратні архітектури.

Однією з ключових переваг використання стандартизованих фреймворків (Arduino) є можливість перенесення коду на різні мікроконтролерні платформи, що дозволяє масштабувати радіоелектронний пристрій залежно від необхідної обчислювальної потужності, енергоефективності тощо.

- а. Створіть новий проєкт, обравши платформу серед доступного переліку (рис. 1.23).



Рис. 1.23. Перелік підтримуваних Wokwi апаратних засобів

- б. Оскільки, залежно від платформи, фізичне розташування портів введення-виведення та внутрішня топологія відрізняються – необхідно оновити схему на одну із наступних (рис. 1.24 – 1.26).

Примітка. При роботі з реальними пристроями, слід звернути особливу увагу на логічні рівні напруги (напр. при переході з 5 В на 3,3 В).

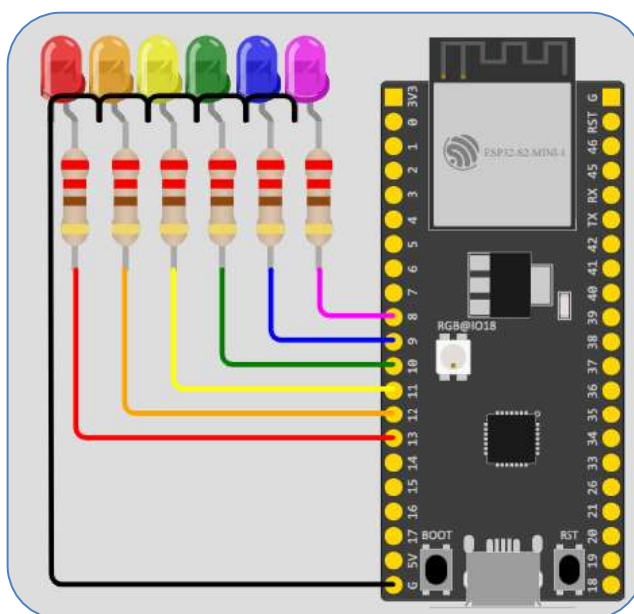


Рис. 1.24. Схема підключення світлодіодів до GPIO-портів мікроконтролера ESP32

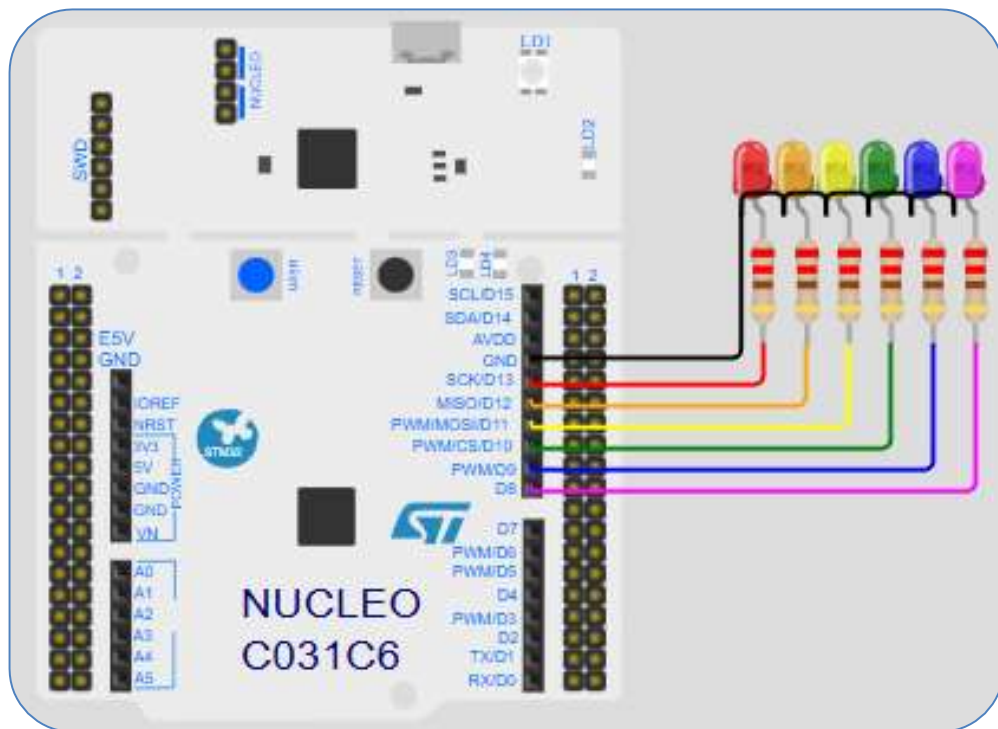


Рис. 1.25. Схема підключення світлодіодів до GPIO-портів мікроконтролера STM32

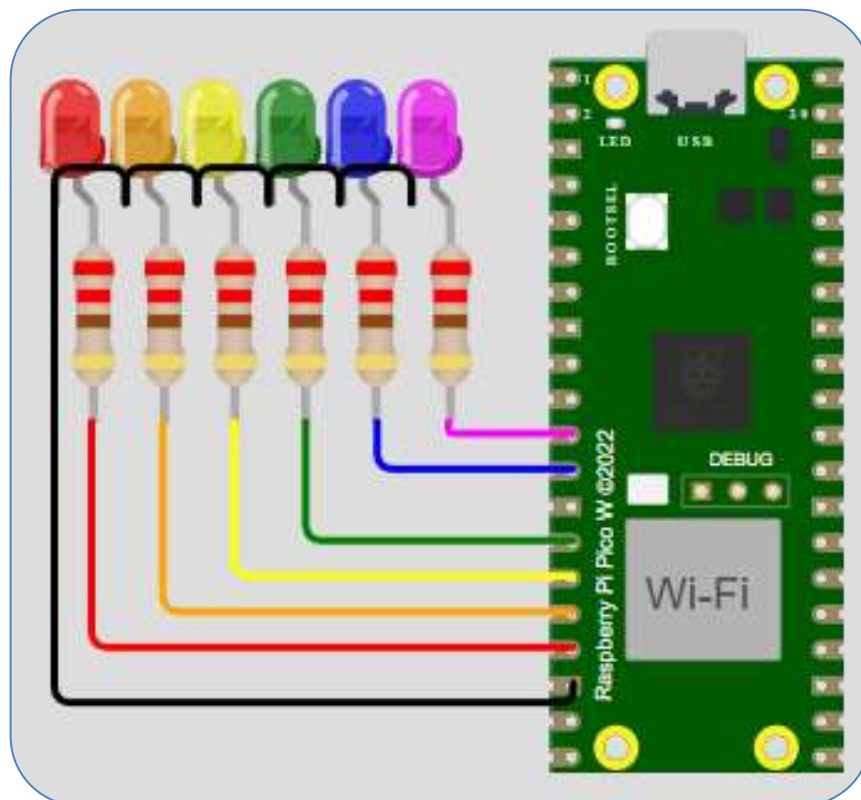


Рис. 1.26. Схема підключення світлодіодів до GPIO-портів мікроконтролера Raspberry Pi Pico

- с. Перевірте працездатність системи та зробіть висновок про переносимість коду написаного з використанням фреймворку Arduino.

Порядок подання звіту

1. Виконайте вхід до курсу на платформі дистанційного навчання «Сікорський».
2. Перейдіть до розділу, що відповідає поточній роботі та відкрийте форму для подання результатів.
3. У полі для текстової відповіді додайте гіперпосилання на збережений проєкт у середовищі Wokwi.

Важливо! Перед відправленням переконайтеся у працездатності проєкту запустивши симуляцію!

4. Збережіть відповідь для внесення проєкту до бази перевірки викладачем.
-

Контрольні запитання

1. У чому полягають основні переваги онлайн-симуляції порівняно з фізичним прототипуванням?
2. Яке призначення використаних у роботі функцій фреймворку Arduino?
3. Яку роль відіграє конфігураційний файл platformio.ini у визначенні параметрів збірки та залежностей проєкту?
4. Які процеси ініціюються кнопкою «Build» / «Upload»?
5. Яка роль файлу diagram.json у симуляторі Wokwi?

ПРАКТИЧНА РОБОТА № 2

ОПЕРАТОРИ ТА ПІДПРОГРАМИ

Мета: Навчитися застосовувати оператори C/C++ та підпрограми для написання структурованого коду та керування електронними модулями.

План проведення практичного заняття

- Опрацювання теоретичних матеріалів.
 - Покрокове виконання завдань.
 - Подання результатів (звіту) на перевірку.
-

Теоретичні відомості

Оператори

Оператори – це символи, які змушують процесор виконувати певні дії над даними. Вони є основою будь-якої програми, оскільки дозволяють виконувати обчислення, порівняння та інші операції.

Типи операторів:

1. **Арифметичні** оператори – виконують стандартні математичні операції:

- + (додавання),
- (віднімання),
- * (множення),
- / (ділення),
- % (остача від ділення).

Приклад: `int result = 10 / 3;` (результат 3).

2. Оператори **присвоєння** – використовуються для надання значення змінній:

- = (присвоєння),
- += (додати та присвоїти),
- = (відняти та присвоїти).

Приклад: `x = 5; x += 2;` (результат 7).

3. Оператори **порівняння** – порівнюють два значення. Результатом завжди є логічне значення (true або false):

== (дорівнює),
 != (не дорівнює),
 > (більше),
 < (менше),
 >= (більше або дорівнює),
 <= (менше або дорівнює).

Приклад: `5 == 5` (результат true).

4. **Логічні** оператори – комбінують або інвертують логічні вирази:

&& (логічне «І») – обидві умови мають бути true,
 || (логічне «АБО») – хоча б одна умова має бути true,
 ! (логічне «НЕ») – інвертує результат.

Приклад: `true && false` (результат false).

5. **Побітові** оператори – виконують операції над окремими бітами чисел:

& (побітове «І»),
 | (побітове «АБО»),
 ^ (побітове «Виключне АБО»),
 ~ (побітове «НЕ»),
 << (зсув вліво),
 >> (зсув вправо).

Приклад: Якщо `int a = 5;` (0101 в двійковій системі), то результатом `a << 1` буде `10` (1010 в двійковій системі).

Підпрограми

При створенні складних програм код розділяють на окремі іменовані блоки – **підпрограми**. Це дозволяє один раз описати складну дію, дати їй «ім'я», а потім – викликати за ним з будь-якого місця програми. Наприклад, коли у коді зустрічається `digitalWrite(13, HIGH)` – відбувається звернення до вже створеного

розробниками Arduino блоку коду з іменем «**digitalWrite**». Всередині якого приховано інструкції для зміни стану біта у регістрі керування портами, що, своєю чергою, змушує апаратну логіку мікроконтролера відкрити відповідний транзисторний ключ та подати напругу на металевий вивід мікросхеми. Використання підпрограми робить код компактним, зрозумілим та легким у налаштуванні.

До підпрограм відносяться:

- **Процедури** – підпрограми, які після виконання певної дії не повертають ніякого результату у точку виклику (тип повернення завжди void).
- **Функції** – підпрограми, що виконують обчислення та повертають результат певного типу у точку виклику.
- **Методи** – це підпрограми (функції/процедури), які визначені всередині класу та призначені для маніпулювання даними цього класу (об'єкта).

Визначення підпрограми – це створення іменованого блоку коду призначеного для виконання конкретної дії. Для цього використовується наступний синтаксис:

```
Тип_результату Ім'я(Параметри) // Заголовок підпрограми
{
    // Тіло підпрограми

    return Результат; // Повернення результату, якщо Тип_результату не void
}
```

Де:

- **Тип_результату** – визначає тип даних, які підпрограма передає назад у точку виклику (наприклад, int для цілих чисел). Якщо підпрограма (процедура) лише виконує дію (як-от подача напруги), але не повертає результат – вказується void;
- **Ім'я** – унікальний ідентифікатор-команда. Використовується для звернення до цього коду з будь-якої частини програми;

- **Параметри** – це перелік даних, необхідних для роботи підпрограми (наприклад, номер GPIO-порту та рівень напруги). Якщо вхідні дані не потрібні, дужки залишаються порожніми;
- **Тіло підпрограми** – область всередині фігурних дужок, де записано послідовність операцій, що будуть виконані після виклику;
- **Повернення результату (return)** – фіксує завершення роботи підпрограми та відправляє обчислене значення назовні. Якщо функція має тип void, цей оператор не використовується.

Виклик підпрограми – це безпосередня команда процесору призупинити виконання основного коду, перейти до іменованого блоку, виконати всі інструкції всередині нього і повернутися назад. Для виклику підпрограми достатньо вказати її ім'я та передати необхідні вхідні дані (аргументи) у дужках:

```
ім'я(значення1, значення2);
```

Значення, що передаються, мають відповідати типам, вказаним при визначенні підпрограми. Наприклад, якщо підпрограма очікує ціле число (int), не варто передавати текст.

Оскільки компіляція відбувається послідовно (з першого рядка до останнього), при спробі виклику підпрограми, що не була визначена в попередніх рядках коду – компілятор видасть помилку **«was not declared in this scope»**, бо ще не «дочитав» до місця, де ця підпрограма визначена.

Оголошення підпрограми – це «обіцянка» компілятору, що повний опис підпрограми з'явиться пізніше. Це дозволяє компілятору зрозуміти як працювати з підпрограмою, ще до того як він побачить її тіло, навіть якщо воно розміщене наприкінці файлу. Складається оголошення лише із заголовка підпрограми та має наступний синтаксис:

```
тип_результату ім'я(параметри);
```

Приклад програми з використанням підпрограм:

```
// Призначення номерів пінів
const int led_blue = 8;
const int led_yellow = 9;

/* ОГОЛОШЕННЯ власної підпрограми
   Повідомляємо компілятору, що підпрограма існує,
   приймає ціле число (int) і нічого не повертає (void) */
void blink_led(int pin_number);

void setup() // ВИЗНАЧЕННЯ підпрограми Arduino, що виконується один раз
при старті
{
    pinMode(led_blue, OUTPUT); // ВИКЛИК підпрограми Arduino pinMode
    pinMode(led_yellow, OUTPUT); // ВИКЛИК підпрограми Arduino pinMode
}

void loop() // ВИЗНАЧЕННЯ підпрограми Arduino, що виконується нескінченно
{
    blink_led(led_blue); // ВИКЛИК власної підпрограми blink_led
    blink_led(led_yellow); // ВИКЛИК власної підпрограми blink_led
}

// ВИЗНАЧЕННЯ власної підпрограми
void blink_led(int pin_number)
{
    digitalWrite(pin_number, HIGH); // ВИКЛИК підпрограми Arduino digitalWrite
    delay(250); // ВИКЛИК підпрограми Arduino delay
    digitalWrite(pin_number, LOW); // ВИКЛИК підпрограми Arduino digitalWrite
    delay(250); // ВИКЛИК підпрограми Arduino delay
}
```

Структура підпрограми **blink_led**:

- **void** – вказує на те, що підпрограма призначена суто для виконання дій і не повертає жодного числового чи логічного результату;
- **blink_led** – ідентифікатор, що використовується для звернення до цього блоку інструкцій;
- **int pin_number** – вхідна змінна цілого типу, яка дозволяє підпрограмі отримати номер конкретного контакту (піна) безпосередньо в момент виклику;
- **Тіло підпрограми** – містить чотири рядки, що є викликами стандартних функцій Arduino.

Наведений приклад демонструє принцип модульності в програмуванні мікроконтролерів. Замість того, щоб в головному циклі (loop) писати ідентичні команди для кожного світлодіода, створено універсальний інструмент – підпрограму **blink_led**. Визначення підпрограми, наведене в кінці прикладу, описує процес блимання світлодіодом. Заздалегідь підпрограма «не знає», яким світлодіодом керувати, а лише має параметр **pin_number**, значення якого отримується безпосередньо в момент виклику. В наведеному коді виклик підпрограми до її визначення відбувається двічі, тому, для уникнення помилок компіляції – підпрограма оголошена на самому початку. В тілі підпрограми виконується виклик інших підпрограм Arduino (**digitalWrite** та **delay**), які на низькому рівні керують мікросхемою. Такий підхід робить код лаконічним і легким для вдосконалення – наприклад, якщо знадобиться змінити частоту блимання для всіх світлодіодів одночасно, це можна буде зробити один раз змінивши тіло підпрограми **blink_led**.

Підпрограма **digitalWrite** [5] керує станом виводів мікроконтролера (подає або знімає напругу з конкретного виводу). Для цього в неї слід передати два аргументи:

```
digitalWrite(Номер_піна, Рівень_напруги);
```

- **Номер_піна** – номер контакту, до якого підключено пристрій (наприклад, світлодіод).
- **Рівень_напруги** – фізичний стан виводу, який може бути: **HIGH** – на піні з'являється напруга, світлодіод світиться, **LOW** – напруга падає до 0 В, світлодіод гасне.

Семисегментний індикатор

Семисегментний індикатор – це модуль для відображення цифрової та символічної інформації (рис. 2.1). Хоча він називається семисегментним, часто він містить 8 світлодіодів (сегментів), сім з яких формують обриси цифри, а восьмий відповідає за десяткову крапку (DP). Характеристики, приклади та способи використання наведені в документації Wokwi [6].

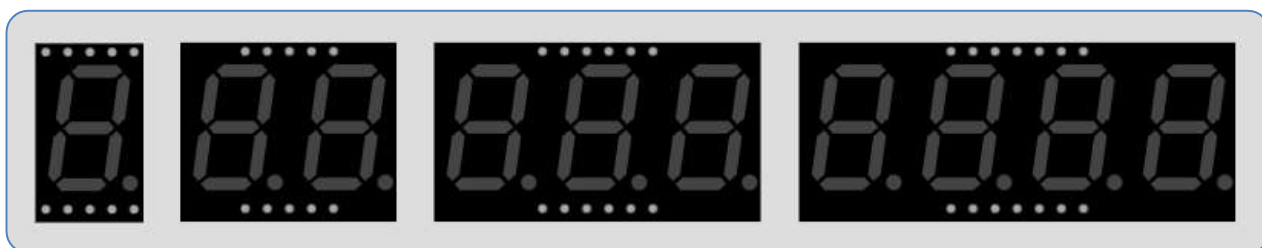


Рис. 2.1. Моделі семисегментних індикаторів, доступні в Wokwi

Більшість стандартних моделей мають 10-пінову структуру. Вісім виводів призначені для формування цифри та крапки, а два інших є загальними точками підключення для всієї групи світлодіодів.

Залежно від внутрішньої конфігурації, сегментні індикатори поділяються на два типи – із загальним анодом та загальним катодом (рис. 2.2), що обов'язково слід враховувати при написанні програмного коду, а саме при реалізації логіки керування рівнями напруги.

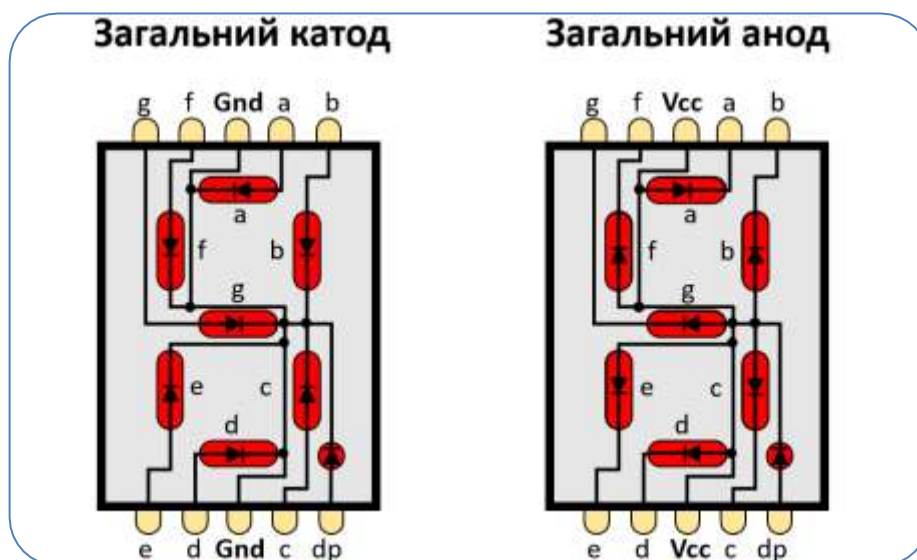


Рис. 2.2. Принципові схеми внутрішнього з'єднання сегментів (загальний катод та анод)

Статична індикація – це метод керування дисплеєм, де кожному сегменту та символу індикатора виділяється окремий, постійно активний контакт для керування. Іншими словами, щоб певний сегмент світився, необхідно постійно (статично) подавати на нього напругу. Засвічування декількох сегментів одночасно дозволяє формувати цифри та літери.

В індикаторі із загальним катодом – всі «мінуси» (катоди) світлодіодів об'єднані та підключаються до землі (**GND**). Щоб сегмент засвітився, на його анод слід подати «плюс», наприклад, за допомогою **digitalWrite(pin_b, HIGH)**.

В індикаторі із загальним анодом усі аноди світлодіодів об'єднані в єдину шину та підключаються до позитивного полюса джерела живлення (**VCC**). Для активації сегмента на катод світлодіода необхідно подати низький логічний рівень: **digitalWrite(pin_b, LOW)**.

Завдання

1. Підключити семисегментний індикатор до плати розробника Arduino для можливості керування кожним його сегментом.
2. Розробити програму для послідовного відображення цифр на індикаторі відповідно до варіанту. За відображення кожної цифри має відповідати окрема підпрограма.

3. Після відображення кожної цифри (x) реалізувати затримку в $x * 100 + 100$ мілісекунд.

Таблиця варіантів

Номер варіанту слід дізнатись у викладача.

Вар.	0	1	2	3	4	5	6	7	8	9
0	—	23425	15617	89081	72374	56758	34536	01203	98796	45647
10	21029	67869	12314	54352	87685	39837	02406	51357	94291	60563
20	17819	25628	43149	76574	08907	35238	92593	61864	47041	83582
30	14916	58250	37139	26428	09507	73175	48642	15913	62760	38431
40	75972	04803	91596	27421	50853	16319	82584	37930	60461	25827
50	41649	12314	08907	37139	51357	39837	08907	45647	83582	14916

Порядок виконання роботи

Крок 1. З'єднати компоненти (рис. 2.3) для можливості керування кожним сегментом індикатора (статична індикація).

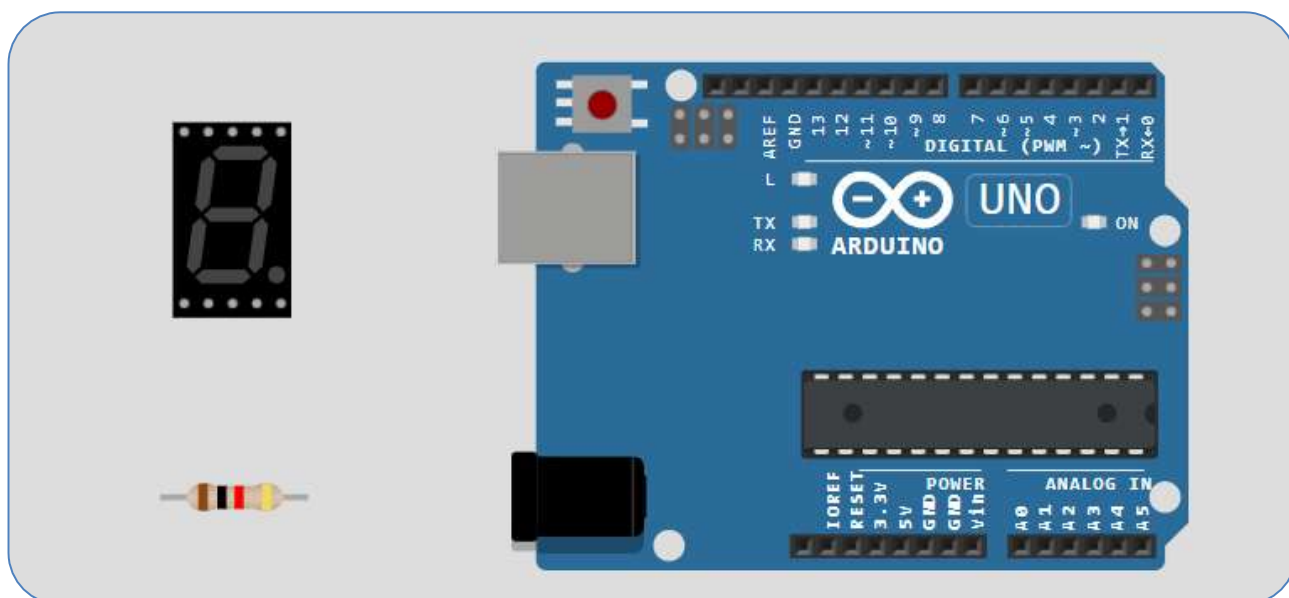


Рис. 2.3. Набір електронних компонентів для виконання роботи

Крок 2. Написати програму з використанням операторів та функцій для реалізації кожного блоку зображеного на блок-схемі (рис. 2.4).

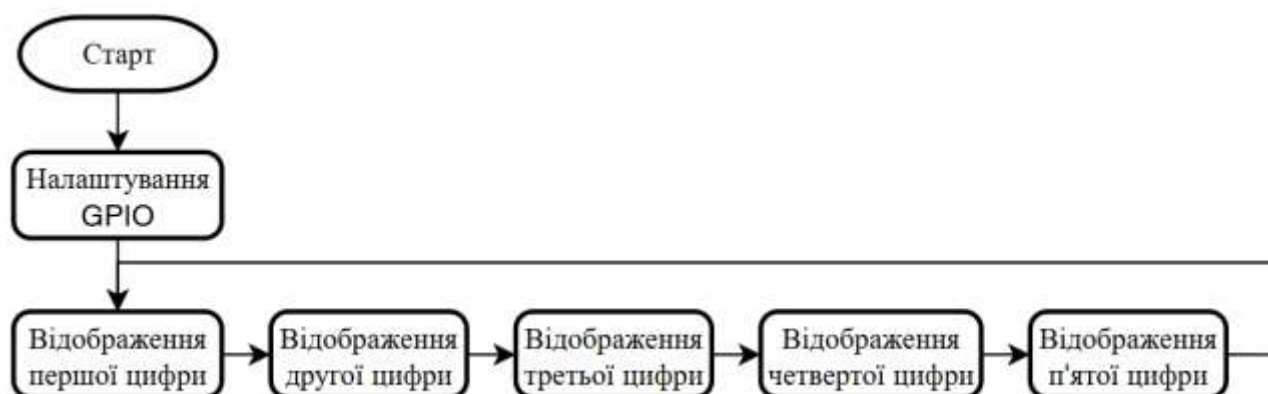


Рис. 2.4. Алгоритм

Контрольні запитання

1. Обґрунтуйте вибір типу даних для кожної змінної/константи в програмі.
2. Які існують оператори та які з них були застосовані?
3. У чому полягає основна мета використання підпрограм у програмуванні?
4. Чим визначення функції відрізняється від оголошення функції?
5. Що таке статична індикація?

ПРАКТИЧНА РОБОТА № 3

КОНТРОЛЬ ПОТОКУ ПРОГРАМИ

Мета: Навчитись використовувати механізми контролю потоку програми та відповідні оператори для реалізації алгоритмів взаємодії з апаратним забезпеченням на основі даних з датчиків та зовнішніх подій введення.

План проведення практичного заняття

- Опрацювання теоретичних матеріалів.
 - Покрокове виконання завдань.
 - Подання результатів (звіту) на перевірку.
-

Теоретичні відомості

Потік виконання – це шлях, за яким процесор опрацьовує інструкції під час функціонування програми. У найпростішому випадку інструкції виконуються лінійно, одна за одною.

Керування потоком виконання – це застосування спеціальних мовних конструкцій для зміни стандартного лінійного порядку обробки команд, що дозволяє створювати складні алгоритми, де певні блоки інструкцій виконуються або пропускаються залежно від динамічних даних. Основними інструментами такого керування є оператори умови, цикли та підпрограми (функції).

Оператори умови – це конструкції розгалуження (**if**, **else**, **switch**), що забезпечують перенаправлення потоку виконання на різні гілки алгоритму. Вибір конкретного шляху базується на результаті обчислення логічного виразу (умови).

Оператор **if**

Оператор умови **if** забезпечує перевірку логічного виразу і дозволяє виконати блок інструкцій лише у разі, якщо цей вираз набуває значення **true** («істина»).

Синтаксис:

```
if (Логічний_вираз)
{
    // блок інструкцій, що виконуватиметься при true
}
```

- **Логічний_вираз** – це умова, сформована за допомогою операторів порівняння або логічних операторів.

Якщо умова хибна (**false**), мікроконтролер ігнорує блок коду у фігурних дужках і переходить до наступної інструкції за їх межами.

Приклад програмної реалізації:

```
int lightLevel = 450; // Поточне значення освітленості (наприклад, з АЦП)

if (lightLevel < 500)
{
    enableNightMode(); // Активувати нічний режим (увімкнути підсвітку)
}
```

У даному фрагменті коду логічним виразом (умовою) є **lightLevel < 500**. Потік виконання програми спрямовується у блок фігурних дужок лише у тому випадку, коли значення змінної менше за 500 одиниць (умова набуває значення **true**). Якщо ж рівень освітленості відповідає нормі або перевищує її (умова **false**), блок інструкцій ігнорується. Таким чином, оператор **if** виступає ключовим інструментом керування потоком виконання, дозволяючи системі адаптувати свою поведінку на основі вхідних параметрів.

Оператор else

У поєднанні з **if**, для створення альтернативної гілки алгоритму, може використовуватись оператор **else**.

Типова конструкція if-else:

```
if (Логічний_вираз)
{
```

```
// Блок інструкцій 1 (виконується при true)
}  
else  
{  
    // Блок інструкцій 2 (виконується при false)  
}
```

- **Блок інструкцій 1** – фрагмент коду, що виконується при істинності логічного виразу.
- **Блок інструкцій 2** – альтернативний фрагмент, що виконується лише тоді, коли результат перевірки умови є хибним.

Приклад програмної реалізації:

```
int lightLevel = 600;  
  
if (lightLevel < 500)  
{  
    // Активація нічного режиму при недостатньому освітленні  
    enableNightMode();  
}  
else  
{  
    // Перехід у денний режим при достатньому рівні світла  
    enableDayMode();  
}
```

У наведеному прикладі, оскільки значення змінної **lightLevel** (600) не задовольняє умову «менше 500», блок **if** буде пропущено. Потік виконання автоматично перейде до блоку **else**, активуючи функцію **enableDayMode()**. Використання конструкції **if-else** гарантує логічну завершеність алгоритму, система завжди перебуватиме в одному з двох визначених станів.

Тернарний оператор

Тернарний оператор – це компактний спосіб реалізації простого розгалуження, який дозволяє замінити багаторядкову конструкцію `if-else` одним виразом. Його доцільно використовувати у випадках, коли необхідно присвоїти значення змінній або виконати коротку дію залежно від умови.

Синтаксис:

```
Логічний_вираз ? Дія_1 : Дія_2;
```

- **Логічний_вираз** – умова, що перевіряється.
- **Дія_1** – виконується, якщо вираз істинний (**true**).
- **Дія_2** – виконується, якщо вираз хибний (**false**).

Припустимо, необхідно вивести в термінал статус рівня освітленості:

```
(lightLevel < 500) ? Serial.print("Низький") : Serial.print("Нормальний");
```

У цьому прикладі, якщо значення `lightLevel` менше `500`, у монітор послідовного порту буде виведено повідомлення «Низький». У протилежному випадку програма виконає інструкцію після двокрапки та виведе «Нормальний».

Оператор множинного вибору `switch`

Оператор **switch** призначений для організації розгалуження потоку виконання на основі порівняння значення певної змінної (селектора) із набором констант. Конструкція на основі **switch** є ефективною альтернативою каскаду операторів **if-else**, коли необхідно обрати один варіант із багатьох.

На відміну від універсальних операторів умови, **switch** працює виключно з цілими числами (**int**, **byte**, **long**) або символами (**char**). Процес виконання починається з обчислення виразу в дужках, після чого потік виконання спрямовується на відповідну мітку **case**, значення якої збігається з результатом обчислення. Обов'язковим елементом кожної гілки є оператор **break**, який припиняє подальшу обробку інструкцій всередині конструкції. Відсутність **break** призведе до послідовного виконання всіх наступних блоків коду, що може стати

причиною логічних помилок. Для обробки випадків, які не передбачені жодною з міток `case`, використовується опціональний блок **default**.

Приклад:

```
int channelNumber = 2; // Номер робочого каналу радіосистеми
switch (channelNumber)
{
    case 1:
        setupChannelOne(); // Налаштування першого каналу
        break;
    case 2:
        setupChannelTwo(); // Налаштування другого каналу
        break;
    case 3:
        setupChannelThree(); // Налаштування третього каналу
        break;
    default:
        handleError(); // Дія у разі некоректного вибору
        break;
}
```

У наведеному прикладі, оскільки змінна **channelNumber** має значення **2**, мікроконтролер ігнорує першу мітку (**case 1**) і переходить безпосередньо до виконання функції **setupChannelTwo()**. Завдяки наявності оператора **break**, після завершення роботи функції потік виконання виходить за межі конструкції **switch**. В інженерній практиці така структура найчастіше використовується для реалізації систем меню, обробки кодів команд або керування режимами роботи складної радіоелектронної апаратури.

Взаємодія з компонентами введення

Тактова кнопка – це комутаційний елемент механічного типу (рис. 3.1), призначений для замикання або розмикання електричного кола з метою формування керуючого сигналу. У контексті мікроконтролерних систем кнопка є

дискретним пристроєм введення, що має два стабільні стани – розімкнутий (логічний «0» або стан високого опору, залежно від схеми підключення) та замкнутий (логічна «1»).

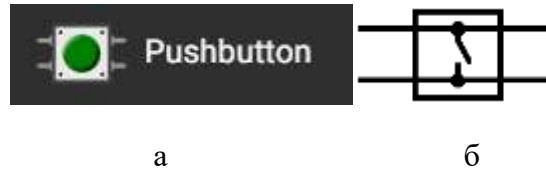


Рис. 3.1. Графічне позначення тактової кнопки в симуляторі (а) та її внутрішня структура контактної групи (б)

Приклад обробки вхідного сигналу:

```
// Призначення ідентифікаторів для пінів введення-виведення
#define PIN_BUTTON 2
#define PIN_INDICATOR 13

// Змінна для фіксації поточного логічного рівня
bool isPressed = false;

void setup()
{
    // Ініціалізація периферії
    pinMode(PIN_BUTTON, INPUT);
    pinMode(PIN_INDICATOR, OUTPUT);
}

void loop()
{
    // Зчитування поточного логічного рівня на піні кнопки
    isPressed = digitalRead(PIN_BUTTON);

    if (isPressed)
    {
        // Увімкнення світлодіода, якщо контакти кнопки замкнені
        digitalWrite(PIN_INDICATOR, HIGH);
    }
}
```

```
}  
else  
{  
    // Вимкнення світлодіода, якщо контакти кнопки розімкнуті  
    digitalWrite(PIN_INDICATOR, LOW);  
}  
  
// Програмна фільтрація перехідних процесів  
delay(150);  
}
```

При використанні механічних комутаційних елементів виникає ефект, відомий як «**вібрація контактів**». Через пружність матеріалів у момент фізичного зіткнення контактів відбувається серія короткочасних неконтрольованих замикань та розмикань, що триває кілька мілісекунд. Мікроконтролер, маючи високу швидкодію, може інтерпретувати цей перехідний процес як послідовність швидких багаторазових натискань, що призводить до некоректної роботи алгоритму. Для нівелювання цього ефекту застосовують різні методи антивідскоку (debouncing). У наведеному прикладі реалізовано найпростіший програмний метод – введення часової затримки **delay()**. Це дозволяє програмі «ігнорувати» стан входу протягом часу, необхідного для завершення перехідних процесів і стабілізації електричного контакту. В професійному проектуванні інтелектуальних систем також застосовують апаратні методи (фільтри) або складніші програмні алгоритми на основі таймерів, без зупинки основного потоку виконання.

Завдання

1. Підключити компоненти до плати розробника Arduino.
2. Розробити програму для зчитування температури з датчика DHT22 та її відображення за допомогою семисегментних індикаторів (статична індикація). За відображення кожної цифри повинна відповідати окрема функція.

3. Реалізувати перевірку та індикацію знаходження температури в допустимих межах при натисканні кнопки (відповідно до варіанту, за допомогою червоного та зеленого світлодіодів).

Таблиця варіантів

Номер варіанту слід дізнатись у викладача.

Вар.	0	1	2	3	4	5	6	7	8	9
0	—	10 - 25	12 - 27	14 - 29	16 - 31	18 - 33	20 - 35	22 - 37	24 - 39	26 - 41
10	28 - 43	30 - 45	32 - 47	34 - 49	36 - 51	38 - 53	40 - 55	42 - 57	44 - 59	46 - 61
20	48 - 63	18 - 33	20 - 35	22 - 37	24 - 39	26 - 41	28 - 43	30 - 45	32 - 47	34 - 49
30	36 - 51	38 - 53	40 - 55	42 - 57	44 - 59	46 - 61	48 - 63	50 - 65	52 - 67	54 - 69
40	56 - 71	15 - 30	17 - 32	19 - 34	21 - 36	23 - 38	25 - 40	27 - 42	29 - 44	31 - 46
50	33 - 48	35 - 50	37 - 52	39 - 54	41 - 56	43 - 58	45 - 60	47 - 62	49 - 64	53 - 68

Порядок виконання роботи

Крок 1. З'єднати компоненти (рис. 3.2) для можливості визначення температури (за допомогою давача DHT22) та її відображення за допомогою семисегментних індикаторів (статична індикація).

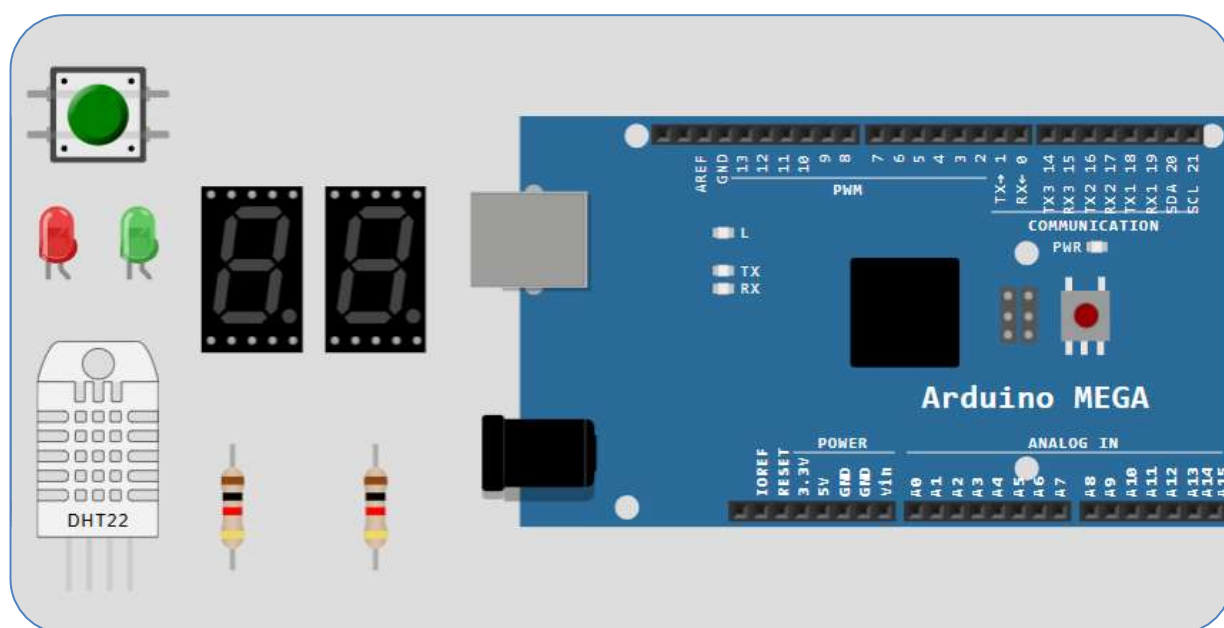


Рис. 3.2. Набір електронних компонентів для виконання роботи

Крок 2. Ознайомитись з особливостями DHT22 [7]. Провести тестування давача, реалізувавши вивід показників температури у вікно монітора послідовного порту:

```
#include <Arduino.h>
#include "DHT.h"

#define DHT_PIN 14           // Номер піна для зчитування даних
#define SENSOR_TYPE DHT22   // Визначення моделі давача

DHT sensor(DHT_PIN, SENSOR_TYPE); // Створення об'єкта для роботи з давачем

void setup()
{
    Serial.begin(115200); // Налаштування швидкості для Serial-інтерфейсу
    sensor.begin();       // Активація сенсора
}

void loop()
{
    // Зчитування поточного значення температури
    float currentTemp = sensor.readTemperature();

    // Вивід результатів вимірювання
    Serial.print("Поточна температура: ");
    Serial.print(currentTemp);
    Serial.println(" °C");

    // Інтервал між опитуваннями (1 секунда)
    delay(1000);
}
```

Крок 3. Написати програму, з використанням операторів та функцій, для реалізації кожного блоку зображеного на схемі (рис. 3.3).

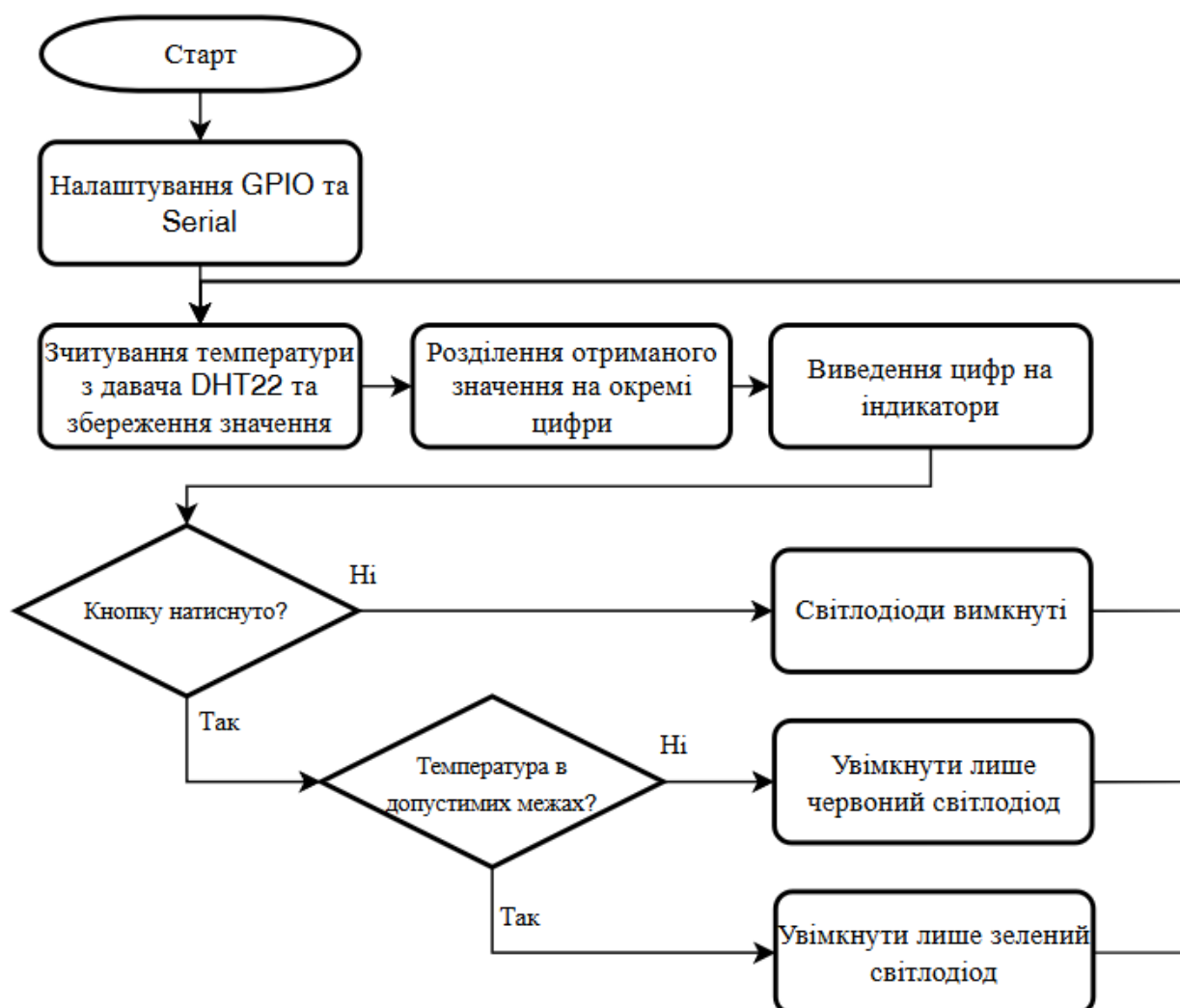


Рис. 3.3. Алгоритм функціонування системи моніторингу температури з візуальною індикацією

Порядок подання звіту

1. Виконайте вхід до курсу на платформі дистанційного навчання «Сікорський».
2. Перейдіть до розділу, що відповідає поточній роботі та відкрийте форму для подання результатів.
3. У полі для текстової відповіді додайте гіперпосилання на збережений проєкт у середовищі Wokwi.

Важливо! Перед відправленням переконайтеся у працездатності проекту запусивши симуляцію!

4. Збережіть відповідь для внесення проєкту до бази перевірки викладачем.
-

Контрольні запитання

1. Як розділити значення змінної (температури) на окремі цифри?
2. Що таке потік виконання програми?
3. Які засоби використовуються для зміни порядку виконання інструкцій у програмі?
4. Для чого використовується оператор **if**?
5. Що таке «умова»? Де вона задається в коді та як зображається на блок-схемі?
6. Коли варто використовувати оператор **switch**?
7. Для чого у функції обробки кнопки потрібна логіка антивідскоку (debouncing)?

ПРАКТИЧНА РОБОТА № 4

ЦИКЛИ, ЧАСОВІ ЗАТРИМКИ ТА ТАЙМЕРИ

Мета: Опанувати принципи побудови програмних циклів, створення часових затримок та конфігурування таймерів мікроконтролера для керування периферійними пристроями, зокрема для реалізації алгоритмів динамічної індикації.

План проведення практичного заняття

- Опрацювання теоретичних матеріалів.
- Покрокове виконання завдань.
- Подання результатів (звіту) на перевірку.

Теоретичні відомості

Цикли

Цикли – це конструкції, призначені для багаторазового виконання певної послідовності інструкцій (ітерацій). Вони забезпечують ефективне керування повторюваними процесами, такими як опитування масивів даних, обробка пакетів даних, генерація сигналів тощо.

Цикл з лічильником (for) – застосовується у випадках, коли кількість ітерацій відома заздалегідь або визначається конкретним діапазоном значень.

Синтаксис:

```
for (Початкове_значення; Логічна_умова; Крок_циклу)
{
    // Код, що повторюється (тіло циклу)
}
```

Цикл for складається з трьох ключових етапів, які керують його роботою:

- **Початкове_значення** – тут зазвичай створюється змінна-лічильник (наприклад, `int i = 0`), від якої починається відлік.

- **Логічна_умова** – перевіряється перед кожним виконанням коду в фігурних дужках. Якщо умова істинна – код виконується. Якщо хибна – цикл завершується.
- **Крок_циклу** – після кожної ітерації значення змінної-лічильника зазвичай збільшується (**i++**) або зменшується, готуючи систему до наступної перевірки умови.

Приклад:

```
for (int i = 0; i < 100; i++)
{
    // Виведення поточного значення «i» та перехід на новий рядок
    Serial.println(i);
}
```

Цикл триватиме доти, доки «i» не набуде значення 100. Оскільки умова $100 < 100$ є хибною (**false**), цикл завершиться, а програма перейде до виконання наступних інструкцій після фігурних дужок. В результаті в терміналі буде надруковано стовпчик чисел: 0, 1, 2, ... , 99.

Цикл з передумовою (while) – забезпечує виконання коду доти, доки задана логічна умова повертає значення **true**. Кількість повторень заздалегідь не обмежена і залежить від динаміки зміни параметрів у тілі циклу або ззовні.

Синтаксис:

```
while (умова)
{
    // Блок інструкцій
}
```

Приклад:

```
int count = 0;           // Ініціалізація змінної-лічильника
while (count < 8) {
    Serial.println(count); // Виведення поточного значення
    count++;              // Модифікація лічильника (інкремент)
}
```

Перед початком кожного проходу (ітерації) мікроконтролер перевіряє вираз у дужках. Якщо умова є істинною – програма заходить всередину фігурних дужок, друкує число та збільшує лічильник на одиницю. Коли після восьмої ітерації змінна-лічильник набуде значення **8**, перевірка умови **8 < 8** поверне **false**. У цей момент цикл припиняє роботу, і програма переходить до наступних інструкцій після блоку **while**.

Якщо умова циклу **while** буде хибною вже при першій перевірці, блок коду всередині дужок не виконається жодного разу. Також слід уважно стежити за зміною лічильника всередині циклу, щоб уникнути «зациклення» – ситуації, коли програма не може вийти з циклу через постійну істинність умови.

Цикл із післяумовою (do-while) – функціонує подібно до циклу **while**, проте має суттєву відмінність у послідовності дій. Перевірка логічної умови відбувається після виконання блоку інструкцій. Це гарантує, що тіло циклу буде виконане принаймні один раз, навіть якщо умова з самого початку є хибною.

Приклад:

```
int count = 0;

do
{
    Serial.println(count); // Виведення поточного значення
    count++;               // Модифікація лічильника
}
while (i < 8);             // Перевірка умови наприкінці
```

На відміну від прикладного програмного забезпечення для персональних комп'ютерів, яке після виконання певної послідовності завдань завершує свою роботу, програми для мікроконтролерів розробляються для безперервного циклічного виконання (постійне опитування сенсорів, оновлення дисплеїв, перевірка кнопок тощо). Коли мікроконтролер вмикається, йому потрібно знати, з якої саме команди починати виконання програми. У класичному C/C++ такою «точкою входу» завжди є функція **main()**. А щоб пристрій працював без зупинки, всередині **main()** створюють нескінченний цикл **while (1)**. Замість цього,

фреймворк Arduino пропонує зручну «обгортку» – підпрограми **setup()** та **loop()**, на основі яких буде автоматично згенеровано **main()**. При цьому завжди можна відмовитися від цієї обгортки й написати звичайну функцію **main()** самостійно:

```
int main()
{
    // тут виконується початкове налаштування (як в Arduino setup())

    while (1)
    {
        // Код головного нескінченного циклу (як Arduino loop())
    }
    return 0; // Недосяжний код (програма ніколи не закривається)
}
```

Часові затримки та таймери

Формування та вимірювання часових інтервалів у мікроконтролері базується на підрахунку тактових імпульсів за допомогою апаратних таймерів/лічильників.

Таймер – це спеціалізований апаратний блок (схема) всередині мікроконтролера, який призначений для точного вимірювання часу та підрахунку зовнішніх або внутрішніх подій. По суті, це програмований лічильник, який працює незалежно від основного циклу виконання програми (**loop**). Він збільшує своє значення з фіксованою швидкістю, яка визначається тактовою частотою мікроконтролера, поділеною на преддільник (prescaler). Коли лічильник досягає заздалегідь визначеного значення (або переповнюється), він може викликати апаратне **переривання**, що змушує процесор миттєво призупинити основний код і перейти до виконання спеціального обробника переривань (ISR).

Таймери застосовуються для реалізації точних затримок та інтервалів, що дозволяють керувати швидкістю виконання коду, генерувати сигнали ШІМ, вимірювати частоти та тривалості імпульсів тощо.

З точки зору програміста, існує два принципово різні **способи програмування** для взаємодії з таймерами, низькорівневий та високорівневий.

Низькорівнева взаємодія – пряме програмування таймера, що дає максимальну точність і контроль, але вимагає глибоких знань апаратної архітектури. Програміст повинен самостійно налаштувати апаратний таймер (встановити регістри мікроконтролера для вибору частоти та режиму роботи) та створити обробник переривань (ISR) (написати спеціальну функцію, яка виконуватиметься щоразу, коли спрацює таймер, забезпечуючи підрахунок часу). Цей підхід використовується в чистому C/C++ без ядра Arduino.

Високорівнева взаємодія – використання готових абстракцій, наприклад функцій `delay()` та `millis()`, що вбудовані в ядро Arduino. Вони автоматично налаштовують апаратний таймер в момент запуску програми та дозволяють програмісту зосередитися на логіці програми.

Затримка – це період часу, протягом якого програма призупиняє або очікує виконання певної дії чи наступної інструкції. Будь-яка затримка у програмуванні мікроконтролерів, навіть найпростіша, насправді є підрахунком, який веде апаратний таймер.

Процес підрахунку:

- 1) **Ініціалізація** – апаратний таймер, який є незалежним лічильником, налаштовується на роботу з певною частотою (визначається тактовою частотою чіпа та преддільником).
- 2) **Встановлення цілі** – програма задає цільове значення для затримки. Наприклад, якщо потрібна затримка 100 мілісекунд, програма обчислює, скільки тактів таймера відповідає цьому часу.
- 3) **Підрахунок** – таймер починає рахувати.
- 4) **Виконання затримки:** для блокуючих затримок – процесор очікує у циклі (виконуючи порожні інструкції), поки таймер не досягне цільового значення; для неблокуючих затримок – процесор продовжує виконувати інший код, а таймер працює у фоновому режимі, викликаючи переривання, коли час вичерпано.

Блокуюча затримка – змушує мікроконтролер призупинити виконання всього коду на вказаний час. Вона реалізується вбудованою функцією Arduino `delay(ms)`. Поки відбувається контрольоване очікування (блокування), мікроконтролер не може виконувати жодних інших інструкцій – він не реагує на кнопки, не зчитує сенсори та не оновлює дисплеї. У складних системах це призводить до пропуску важливих подій.

Приклад (блокування):

```
void loop()
{
    digitalWrite(LED_PIN, HIGH);
    delay(1000); // Виклик функції призводить до БЛОКУВАННЯ на 1 секунду!

    digitalWrite(LED_PIN, LOW);
    delay(1000); // Виклик функції призводить до БЛОКУВАННЯ на 1 секунду!

    // Тільки через ~2 секунди програма зможе перевірити стан кнопки
    if (digitalRead(BUTTON_PIN) == HIGH)
    {
        // ...
    }
}
```

Неблокуюча затримка не зупиняє програму, що дозволяє мікроконтролеру виконувати кілька завдань «одночасно». У фреймворку Arduino вона реалізується за допомогою вбудованої функції `millis()`, яка повертає кількість мілісекунд, що минули з моменту запуску мікроконтролера. Замість зупинки виконання програми – в коді постійно перевіряється чи минув потрібний інтервал часу для виконання певної інструкції.

Приклад:

```
long previousTime = 0;
long interval = 1000;
```

```

void loop()
{
    // 1. Неблокуюча дія (блмання світлодіода)
    if (millis() - previousTime >= interval)
    {
        // Час минув – оновлюємо час та виконуємо дію
        previousTime = millis();
        digitalWrite(LED_PIN, !digitalRead(LED_PIN));
    }
    // 2. Паралельна дія (перевірка кнопки)
    // Цей код виконується постійно, не чекаючи, поки світлодіод блмне
    if (digitalRead(BUTTON_PIN) == HIGH)
    {
        // ...
    }
}

```

У циклі `loop()` постійно перевіряється, чи минув потрібний інтервал, використовуючи формулу: **Поточний час – Час останньої події $\geq$ Інтервал**. Якщо умова виконана, одразу оновлюється `previousTime` до поточного значення `millis()` та виконується потрібна дія (зміна стану світлодіода). Якщо умова не виконується, програма продовжує перевіряти стан кнопки.

Цей підхід необхідно застосовувати завжди, коли треба одночасно виконувати дві або більше незалежних дій (наприклад, зчитувати температуру, відображати її на дисплеї і перевіряти кнопку).

Динамічна індикація

Більшість комерційних багаторозрядних індикаторів спроектовані для роботи в режимі динамічної індикації, що значно зменшує кількість необхідних контактів та провідників, знижує енергоспоживання та спрощує апаратну реалізацію пристрою.

Динамічна індикація – це метод керування багаторозрядними світлодіодними панелями, що базується на принципі часового розділення каналів. На відміну від статичного методу, де кожен сегмент підключений до окремого порту мікроконтролера, динамічний підхід передбачає використання спільної шини даних для всіх розрядів.

Наприклад, кожен із розрядів чотирирозрядного 7-сегментного індикатора містить 8 елементів (7 сегментів та десяткову крапку). За статичної індикації для підключення знадобилося б 32 контакти. Для динамічної індикації знадобиться лише 12 ліній, з яких 8 утворюють спільну шину сегментів, а 4 відповідають за вибір активного розряду. В будь-який момент часу активним є лише один розряд, що призводить до зниження загального пікового споживання струму порівняно зі статичною індикацією, де всі світлодіоди світяться одночасно.

Процес відображення інформації реалізується через циклічне виконання наступних етапів:

- 1) **Селекція розряду** – програма активує керуючий вихід лише одного розряду (позиції) індикатора.
- 2) **Кодування символу** – на спільну шину сегментів подається двійковий код цифри, призначеної саме для цієї позиції.
- 3) **Експозиція** – розряд залишається активним протягом короткого проміжку часу (зазвичай 2–5 мс). Цього достатньо для збудження люмінофору світлодіода, але замало для фіксації фази вимкнення оком.
- 4) **Інкремент позиції** – поточний розряд деактивується, і цикл миттєво повторюється для наступного розряду.

Цей процес відбувається настільки швидко, що людське око не встигає помітити вимкнення світлодіодів. Завдяки ефекту персистентності зору (persistence of vision), мозок сприймає ці послідовні спалахи як постійне і одночасне світіння всіх цифр.

Динамічна індикація неможлива без двох ключових елементів: **циклу (for/while)** та **точного контролю часу**.

Використання циклу (**for/while**) забезпечує швидкий і послідовний перебір усіх розрядів (наприклад, від 1-го до 4-го) і повторює цю послідовність нескінченну кількість разів.

Швидкість перемикання (частота оновлення) повинна бути високою (зазвичай, вище 50 Гц для всього циклу). Для цього потрібні короткі затримки (**delay(2)**) або точний контроль часу через **millis()** між активацією кожного розряду.

Завдання

1. Розробити функцію **updateDisplay(int number)**, для відображення на дисплеї будь-якої можливої комбінації цифр.
2. Реалізувати роботу пристрою у двох режимах (забезпечити миттєве перемикання режимів при натисканні кнопки):
 - a. Відображення в реальному часі кількості секунд, що минули з моменту запуску програми. Таймер працює в штатному режимі.
 - b. Відображення номера варіанта на дисплеї. При переході в цей режим відлік часу повинен зупинятись («заморожуватись»), а при поверненні до попереднього режиму – продовжуватись з значення, на якому відбулася зупинка.

Примітка. Програма повинна моментально реагувати на натискання кнопки! Уникайте блокуючих затримок!

Додаткове завдання 1. Додати звуковий сигнал при натисканні кнопки (використовуючи неблокуючі затримки при генерації звуку).

Додаткове завдання 2. В другому режимі відображати кількість секунд які були витрачені на демонстрацію номеру варіанту.

Важливо! Бали за додаткові завдання нараховуються за умови вчасного подання звіту.

Порядок виконання роботи

Крок 1. Зібрати електричну схему згідно з рисунком 4.1:

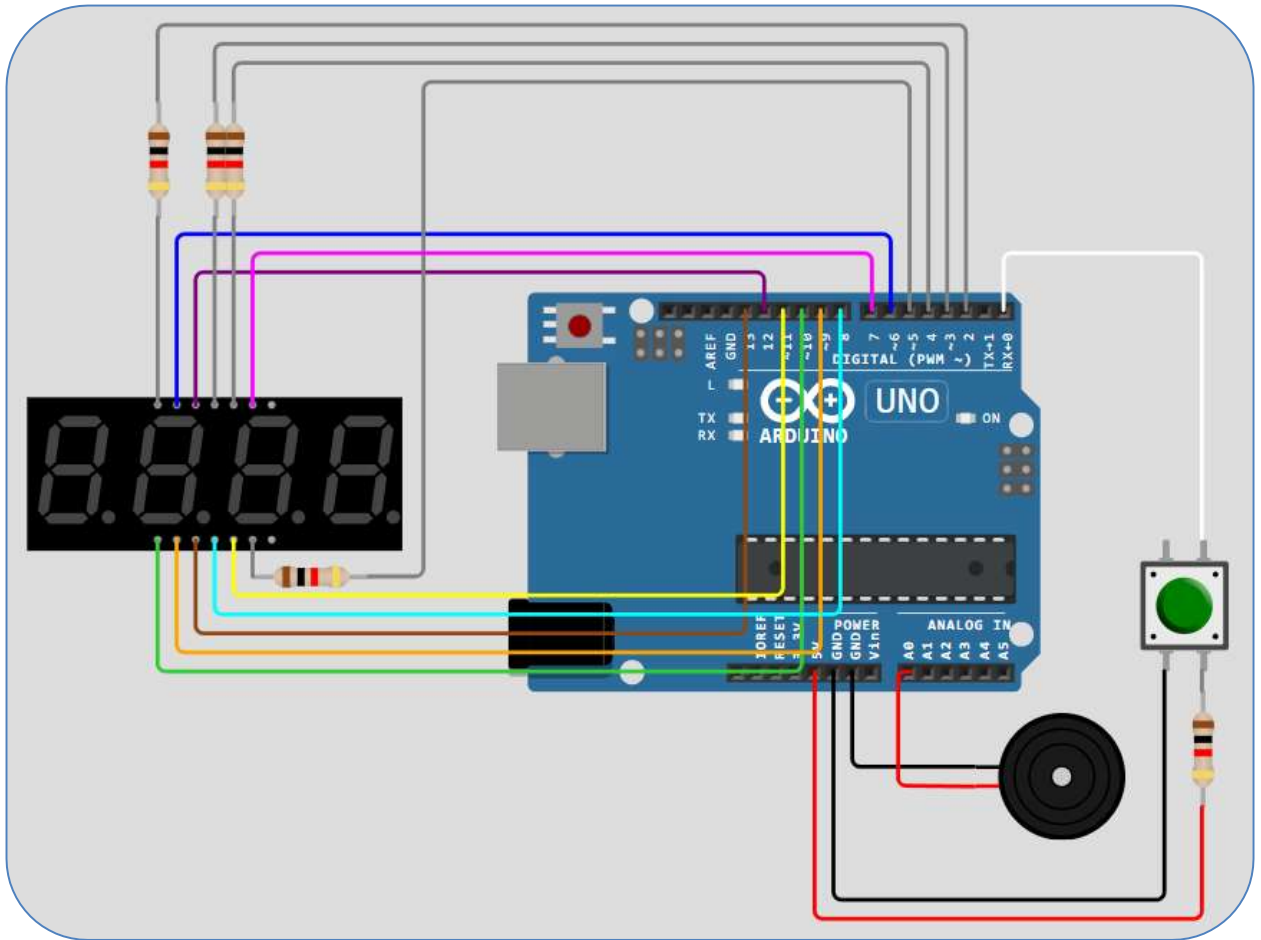


Рис. 4.1. Схема підключення багаторозрядного індикатора

- а. Підключіть чотирирозрядний семисегментний індикатор, кнопку та п'єзодинамік до плати Arduino.
- б. Налаштуйте GPIO у функції **setup()** або на початку функції **main()** (для додаткового завдання):
 - налаштуйте всі піни сегментів та піни розрядів як виходи (**OUTPUT**);
 - налаштуйте пін кнопки як вхід (**INPUT_PULLUP**);
 - ініціалізуйте **Serial** (за потреби, для налагодження).

Крок 2. Розробити функцію керування дисплеєм **updateDisplay(int number)**:

- a. Створіть 10 окремих функцій або одну велику функцію/масив, яка встановлює стани (**HIGH/LOW**) на 8 пінах сегментів, щоб відобразити одну цифру (0–9).
- b. Створіть функцію **updateDisplay(int number)**, яка приймає ціле чотиризначне число (наприклад, 1234). Усередині функції розділіть число на чотири окремі цифри (тисячі, сотні, десятки, одиниці) за допомогою арифметичних операторів (/ та %). Використовуйте оператор циклу (**for** або **while**) для ітеративного перебору від 1-го до 4-го розряду. Усередині циклу:
 - активуйте один розряд (увімкніть його пін);
 - викличте функцію, що відображає відповідну цифру;
 - реалізуйте мікрозатримки;
 - деактивуйте розряд (вимкніть його пін).

Крок 3. Реалізація секундоміра. Можна скористатись одним з двох варіантів:

- 1) Використайте **millis()** напряду. Для цього, в основному циклі **loop()** при виклику функції відображення використовуйте **updateDisplay(millis()/1000);**.
- 2) Оголосіть глобальні змінні: **previousTime** (зберігає час останнього оновлення); **interval** (інтервал оновлення); **secondsCount** (внутрішня змінна для зберігання кількості секунд). Створіть функцію **runStopwatch()**, яка не приймає аргументів. Всередині функції використайте неблокуючий підхід з **millis()** для збільшення значення змінної **secondsCount** на одиницю щосекунди.

Крок 4. Керування потоком та режимами:

- a. Оголосіть глобальну змінну для збереження поточного режиму **modeState** (0 – Секундомір, 1 – Номер варіанту).
- b. Розробіть функцію **processButton()**:
 - реалізуйте логіку антивідскоку;
 - якщо кнопка натиснута – перемкніть змінну **modeState** між 0 і 1.
- c. Скомпонуйте всі функції в головному нескінченному циклі:

- розпочніть з виклику `processButton()` (для миттєвої реакції на натискання кнопки і зміни режиму);
- використовуючи оператор умови (`if/else`) перевіряйте `modeState`: якщо `modeState == 1` – викличте `updateDisplay(X)`, де `X` — номер варіанту (наприклад, 0001); якщо `modeState == 0` – викличте `updateDisplay(secondsCount)`, де `secondsCount` – виміряна кількість секунд під час демонстрації номеру варіанту.

Порядок подання звіту

1. Виконайте вхід до курсу на платформі дистанційного навчання «Сікорський».
2. Перейдіть до розділу, що відповідає поточній роботі та відкрийте форму для подання результатів.
3. У полі для текстової відповіді додайте гіперпосилання на збережений проєкт у середовищі Wokwi.

Важливо! Перед відправленням переконайтеся у працездатності проєкту запусивши симуляцію!

4. Збережіть відповідь для внесення проєкту до бази перевірки викладачем.

Контрольні запитання

1. Який оператор циклу необхідний для реалізації динамічної індикації?
2. Чим відрізняються статичний та динамічний методи керування індикаторами?
3. Який апаратний блок забезпечує точний відлік часу з моменту запуску плати?
4. Що таке «блокуюча затримка» і чому її слід уникати?

5. Що є основою багатозадачності та дає можливість миттєво реагувати на натискання кнопки навіть в процесі динамічної індикації?
6. Яка функція використовується для створення неблокуючої затримки?
7. Як зберігати та змінювати режими роботи пристрою?

ПРАКТИЧНА РОБОТА № 5

КОМУНІКАЦІЯ З ПЕРИФЕРІЙНИМИ ПРИСТРОЯМИ ЗА ПРОТОКОЛАМИ UART, SPI ТА I2C

Мета: Навчитися підключати периферійні пристрої до мікроконтролера та налаштовувати обмін даними за допомогою протоколів UART, SPI та I2C.

План проведення практичного заняття

- Опрацювання теоретичних матеріалів.
 - Покрокове виконання завдань.
 - Подання результатів (звіту) на перевірку.
-

Теоретичні відомості

Універсальний асинхронний приймач-передавач (UART, Universal Asynchronous Receiver-Transmitter)

UART як апаратний модуль – це фізична частина мікроконтролера (блок транзисторів і логічних вентилів), яка бере 8 біт даних, які лежать поруч у пам'яті (паралельно), і «виштовхує» їх по черзі в один дріт (послідовно). Апаратний модуль має власний буфер (пам'ять) і регістри, тому здатен обробляти вхідні та вихідні потоки автономно, не задіюючи ресурси центрального процесора.

UART як протокол – це правила спілкування, за якими працює цей апаратний модуль.

Обмін даними відбувається через дві незалежні лінії, що дозволяє реалізувати повнодуплексний зв'язок: TX – для відправлення інформаційних бітів; RX – для зчитування отриманих сигналів. На платформі Arduino UNO (рис. 5.1) це виводи 0 (RX) та 1 (TX). Їх не слід задіювати для підключення іншої периферії під час активного сеансу зв'язку, щоб уникнути електричних конфліктів та втрати пакетів.



Рис. 5.1. UART-піни на платах розробника Arduino та ESP32

Особливістю UART є відсутність зовнішнього сигналу синхронізації. Координація між передавачем та приймачем досягається виключно шляхом попереднього узгодження часових параметрів (Baud Rate) та структури кадру (рис. 5.2).

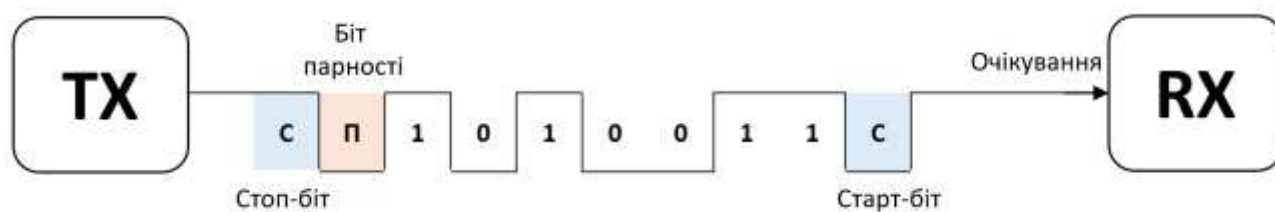


Рис. 5.2. Процес передачі кадру

Апаратний модуль автоматично формує інформаційний пакет, що складається з таких елементів:

- старт-біт – короткочасне зниження рівня сигналу, що «пробуджує» приймач;
- корисне навантаження (найчастіше 8 біт);
- опціональний біт парності для виявлення помилок;
- стоп-біт – фіксація високого рівня на лінії, що сигналізує про успішне завершення передачі.

Клас Serial

Для взаємодії з модулем UART у середовищі Arduino використовується вбудований клас Serial [5]. Він надає набір методів для конфігурування порту, а також для відправлення та отримання даних, включаючи наступні:

- **Serial.begin(*швидкість*)** – активує послідовний порт і встановлює швидкість передачі.
- **Serial.print(*дані*)** – відправляє дані у вигляді тексту. Тобто спочатку дані перетворюються у символи, а лише потім потрапляють у вихідний буфер UART. Наприклад, при відправці **Serial.print(123)**, мікроконтролер не відправляє число **123** як один байт, а відправляє три окремі байти: код символу **'1'** (49 в ASCII); код символу **'2'** (50 в ASCII); код символу **'3'** (51 в ASCII).
- **Serial.println(*дані*)** – аналогічний до **print**, але автоматично додає символи повернення каретки та переведення рядка (**\r\n**), що забезпечує перехід на новий рядок у моніторі порту.
- **Serial.write(*дані*)** – відправляє «сирі» бінарні дані (байти). Використовується для міжмашинної взаємодії, де важлива швидкість, а не читабельність тексту.
- **Serial.available()** – повертає кількість байтів, що вже надійшли в буфер і готові до зчитування. Це дозволяє уникнути зчитування «порожнього» буфера.
- **Serial.read()** – вилучає та повертає перший доступний байт із буфера. Якщо буфер порожній, метод поверне **-1**.
- **Serial.readStringUntil(*символ-розділювач*)** – зчитує весь потік даних у змінну типу **String**, доки не зустрине символ-розділювач (наприклад, **'\n'**).

Наведений нижче приклад демонструє базовий механізм взаємодії між мікроконтролером та персональним комп'ютером. Програма очікує на

надходження символів, зчитує їх із вхідного буфера та повертає назад у текстовому форматі:

```
char incomingChar = 0; // Змінна для зберігання отриманого символу

void setup()
{
    // Ініціалізація UART на швидкості 9600 бод
    Serial.begin(9600);
    Serial.println("Система готова. Очікування вводу символів...");
}

void loop()
{
    // Перевірка наявності байтів у вхідному черзі (буфері)
    if (Serial.available() > 0)
    {
        // Зчитування найпершого байта з черги
        incomingChar = Serial.read();

        // Формування звіту про отримання
        Serial.print("Отримано символ: ");

        // Відправка "сирого" байта (символу) назад у термінал
        Serial.write(incomingChar);

        // Перехід на новий рядок для наступного повідомлення
        Serial.println();
    }
}
```

У блоці **setup()** налаштовується зв'язок на швидкості 9600 бод (стандартне значення, яке має збігатися з налаштуваннями монітора порту). Метод **Serial.available()** постійно перевіряє, чи надійшли дані від комп'ютера. Команда

Serial.read() вилучає символ із буфера, після чого він відправляється вже мікроконтролером за допомогою **Serial.write()**.

Оскільки середовище Wokwi орієнтоване на симуляцію одного мікроконтролера в межах одного проєкту, для дослідження обміну даними між двома пристроями через лінії TX/RX рекомендується скористатися матеріалами з офіційної документації Arduino [8].

Послідовний периферійний інтерфейс (SPI, Serial Peripheral Interface)

SPI – один із найшвидших протоколів послідовного зв'язку, що працює в повнодуплексному режимі (Full Duplex). На відміну від UART, SPI є синхронним, тобто обмін даними чітко координується спільним тактовим сигналом між двома типами пристроїв – головним та другорядними.

Головний (Main/Master) – пристрій, що генерує тактовий сигнал та ініціює обмін даними.

Другорядні (Secondary/Slave) пристрої приймають тактовий сигнал та відповідають на запити головного.

Для комунікації по SPI необхідно використовувати чотири фізичні лінії:

- **SCK** (Serial Clock) – лінія тактування. Головний пристрій подає на неї імпульси, які визначають швидкість передачі кожного біта.
- **MOSI** (**M**ain **O**ut **S**econdary **I**n) – лінія передачі даних від головного пристрою до другорядних.
- **MISO** (**M**ain **I**n **S**econdary **O**ut) – лінія передачі даних від другорядних пристроїв до головного.
- **SS** (Secondary Select) – лінія вибору конкретного другорядного пристрою. Від головного пристрою до кожного другорядного підключається окрема лінія SS. Коли на цій лінії встановлюється низький логічний рівень, другорядний пристрій «розуміє», що головний звертається саме до нього.

Для визначення фізичних виводів мікроконтролера, що призначені для роботи з інтерфейсом SPI, можна звернутися до технічної документації або скористатися програмним методом. Номери виводів для інтерфейсу SPI визначені в конфігураційних файлах за допомогою директив MOSI, MISO, SCK та SS. Їх можна вивести в серійний монітор:

```
#include <Arduino.h>

void setup()
{
    Serial.begin(9600);

    Serial.println(MOSI);
    Serial.println(MISO);
    Serial.println(SCK);
    Serial.println(SS);
}

void loop()
{
}
```

Розташування даних виводів на платах Arduino UNO та ESP32 наведено на рисунку 5.3. Слід зауважити, що на відміну від Arduino UNO, де піни SPI є фіксованими, у контролері ESP32 апаратний модуль SPI може бути перепризначений на інші виводи за допомогою програмних налаштувань, проте за замовчуванням використовуються вказані значення.

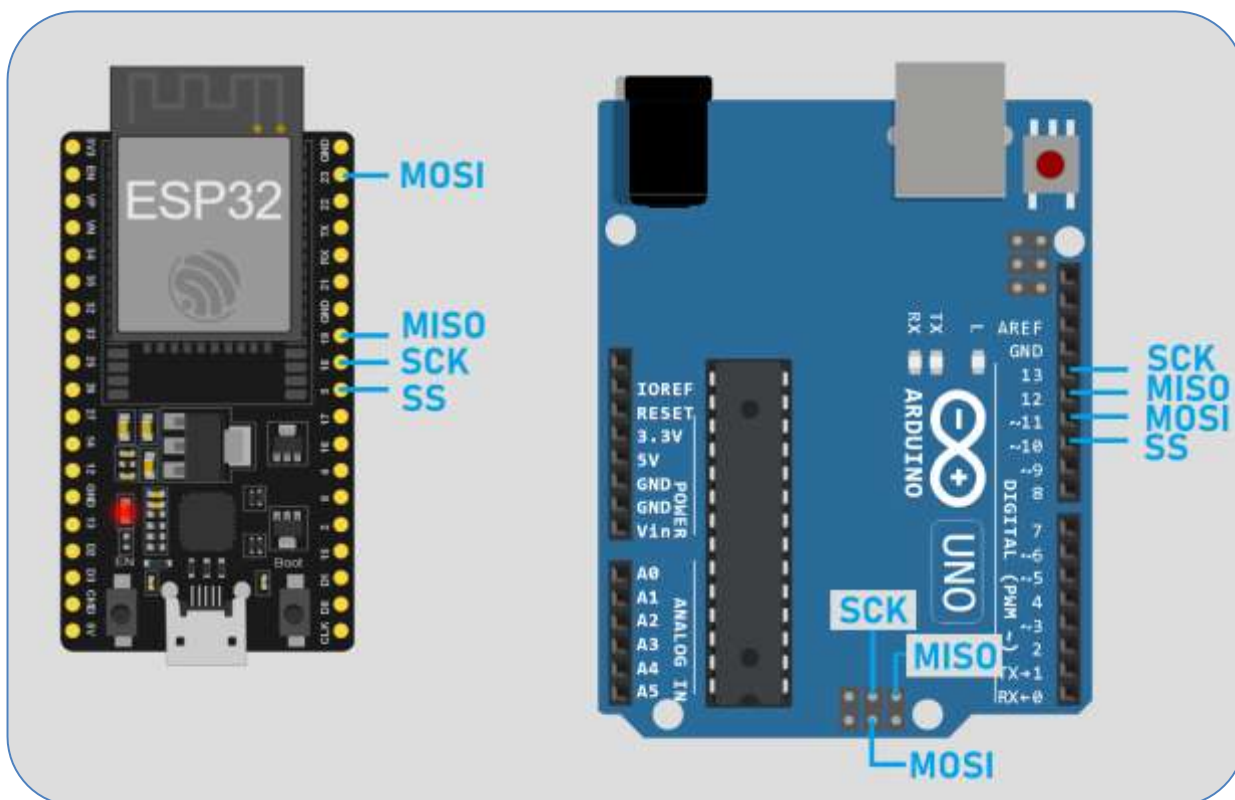


Рис. 5.3. Розташування виводів інтерфейсу SPI на платах Arduino UNO та ESP32

Для організації обміну даними між мікроконтролером (Main) та периферійним пристроєм (Secondary) необхідно дотримуватися наступної послідовності дій:

- 1) З'єднати однойменні лінії MOSI, MISO та SCK обох пристроїв. Лінію SS підключити до будь-якого вільного цифрового піна «головного».
- 2) Підключити стандартну бібліотеку SPI.h, для роботи з апаратним модулем:

```
#include <SPI.h>
```

- 3) Пін, що виконує роль SS, обов'язково має бути переведений у режим виходу:

```
pinMode(SS_PIN, OUTPUT);
```

- 4) Оскільки периферійні пристрої мають фіксовані заводські характеристики, мікроконтролер повинен підлаштувати роботу свого апаратного модуля під «паспортні дані» конкретного другорядного пристрою. Для цього

використовується метод **SPI.beginTransaction(SPISettings(Speed, BitOrder, Data Mode))**, який приймає три параметри:

- **Speed** – гранична частота тактування в Гц (наприклад, 14000000), яку підтримує периферійний пристрій.
- **Bit Order** – визначає, який біт мікроконтролер має надсилати першим – старший (**MSBFIRST**) чи молодший (**LSBFIRST**).
- **Data Mode** – один із чотирьох режимів SPI (**SPI_MODE0** – **SPI_MODE3**) [5], що вказує, у який саме момент фіксуватиметься значення біта – коли тактовий сигнал тільки починає змінюватися чи коли він повертається у вихідний стан.

- 5) Для ініціації сеансу з конкретним другорядним пристроєм, головний встановлює на відповідній лінії SS низький логічний рівень:

```
digitalWrite(SS_PIN, LOW);
```

- 6) Процес передачі реалізується за допомогою методу **SPI.transfer()**. За кожен виклик він відправляє один байт від майстра до підлеглого і одночасно з цим обробляє байт, отриманий від підлеглого у відповідь:

```
SPI.transfer(0b00100111);
```

- 7) Після завершення передачі головний пристрій деактивує другорядний, повертаючи пін SS у високий стан за допомогою:

```
digitalWrite(SS_PIN, HIGH);
```

- 8) Завершення сеансу зв'язку виконується за допомогою методу:

```
SPI.endTransaction();
```

Він звільняє шину та дозволяє мікроконтролеру переналаштувати апаратний модуль під параметри наступного пристрою (наприклад, з швидкої SD-карти на повільний датчик), а також знімає блокування переривань, які могли виникнути під час передачі даних.

- 9) У випадках, коли подальше використання інтерфейсу SPI не планується, викликається метод, який повністю деактивує апаратний модуль SPI всередині мікроконтролера, повертаючи задіяні виводи (MOSI, MISO, SCK) у стан звичайних пінів введення-виведення загального призначення:

```
SPI.end();
```

Прикладом застосування SPI є підключення 8-розрядного зсувного регістра 74HC595 до мікроконтролера Arduino UNO (рис. 5.4). Ця інтегральна схема дозволяє реалізувати перетворення послідовного коду в паралельний, що, наприклад, може використовуватись для керування світлодіодами за умови дефіциту вільних ліній вводу-виводу мікроконтролера.

Згідно зі специфікацією 74HC595, обмін даними здійснюється через наступні виводи:

- **DS** (Serial Data Input) – вхід послідовних даних, підключається до лінії MOSI (пін 11 Arduino UNO);
- **SHCP** (Shift Register Clock Input) – вхід тактування зсувного регістра, підключається до лінії SCK (пін 13 Arduino UNO);
- **STCP** (Storage Register Clock Input) – підключається до лінії SS.

Вивід **OE** (Output Enable) з'єднується із загальною землею (GND) для активації вихідних буферів, а вивід **MR** (Master Reset) підключається до шини живлення (VCC) для запобігання асинхронному скиданню даних.

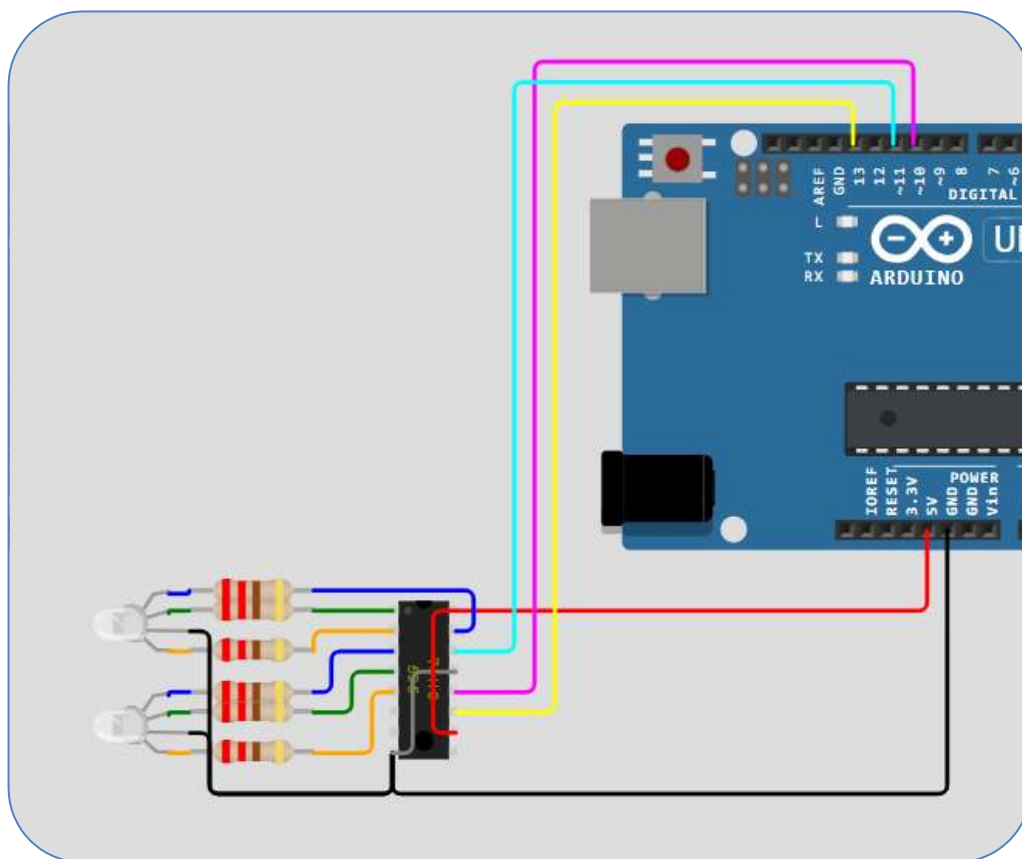


Рис. 5.4. Схема підключення 74HC595 до Arduino UNO

Для прямого керування одним RGB-світлодіодом потрібно 3 цифрових виходи мікроконтролера. Кожен новий світлодіод «забиратиме» ще по 3 додаткових піни, швидко вичерпуючи їх кількість. В наведеному прикладі, з використанням зсувного регістра та SPI, задіяно також 3 піни (MOSI, SCK, SS). Однак, якщо знадобиться додати ще декілька IC 74HC595 (для підключення нових світлодіодів), лінії MOSI та SCK залишаться спільними для всіх. Знадобиться лише додати відповідні лінії SS до кожної нової IC або налаштувати їх роботу в каскадному режимі.

IC 74HC595 дозволяє керувати станом восьми вихідних ліній (Q0–Q7), використовуючи лише один байт даних (8 біт). Кожен біт у переданому байті напряму відповідає за стан конкретного виходу: логічна «1» вмикає відповідний кристал (червоний, зелений або синій), а логічний «0» – вимикає його. Наприклад, надіславши байт, у якому встановлені біти для синього та зеленого каналів, можна отримати колір Aqua. Увімкнення всіх каналів (надсилання **0b11111111**) призведе

до світіння усіх підключених RGB-світлодіодів (з спільним катодом) білим кольором.

```
#include <Arduino.h>
#include <SPI.h>

// Визначення піна вибору пристрою (Secondary Select)
const int CHIP1_PIN = 10;

void setup()
{
    // Налаштування піна SS на вихід та переведення у пасивний стан (HIGH)
    pinMode(CHIP1_PIN, OUTPUT);
    digitalWrite(CHIP1_PIN, HIGH);

    // Ініціалізація апаратного інтерфейсу
    SPI.begin();
}

void loop()
{
    // Налаштування параметрів конкретної сесії обміну даними
    SPI.beginTransaction(SPISettings(14000000, LSBFIRST, SPI_MODE0));

    // Активація кольору Aqua RGB#1 (синій + зелений канали)
    digitalWrite(CHIP1_PIN, LOW);    // Початок передачі
    SPI.transfer(0b0000011);         // Запис бітової маски
    digitalWrite(CHIP1_PIN, HIGH);   // Деактивація другорядного пристрою

    delay(500);

    // Деактивація всіх каналів (вимкнення світлодіода)
    digitalWrite(CHIP1_PIN, LOW);    // Початок передачі
    SPI.transfer(0b00000000);         // Запис маски
    digitalWrite(CHIP1_PIN, HIGH);   // Деактивація другорядного пристрою
}
```

```
// Завершення транзакції для звільнення шини
SPI.endTransaction();

delay(500);
}
```

Зв'язок між інтегральними схемами (I2C, Inter-Integrated Circuit)

I2C – це синхронний двопровідний послідовний інтерфейс, призначений для зв'язку між інтегральними схемами всередині електронних пристроїв. Фізично I2C реалізований за топологією «загальна шина». Усі пристрої (головні та другорядні) паралельно підключені до двох спільних ліній – SDA та SCL (рис. 5.5):

- SDA (Serial Data Line) – лінія для двоспрямованої передачі даних між головним та другорядними пристроями.
- SCL (Serial Clock Line) – лінія тактового сигналу, що забезпечує синхронізацію передачі бітів.

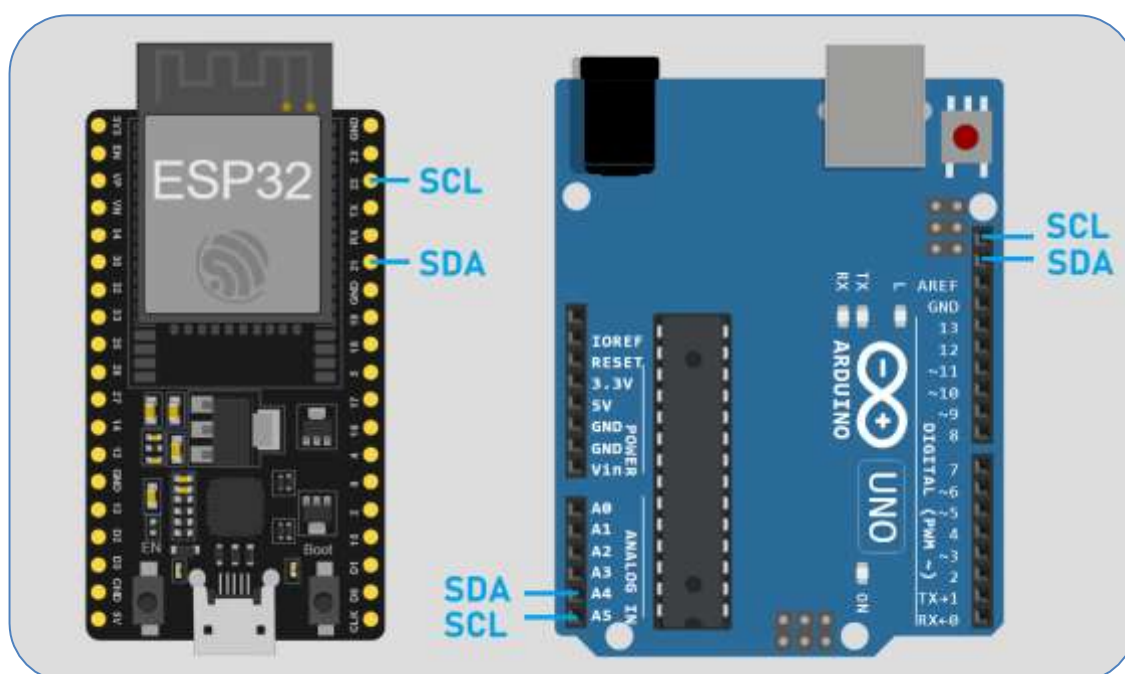


Рис. 5.5. Розташування виводів інтерфейсу I2C на платах ESP32 та Arduino UNO

Оскільки лінії зв'язку спільні для всіх, вибір конкретного отримувача базується на адресному просторі. Кожен Secondary-пристрій має власну жорстко

задану або налаштовувану адресу (зазвичай 7 біт). При встановленні зв'язку, Main-пристрій відправляє потрібну адресу по SDA, усім іншим пристроям, а відповідає йому лише той, чия адреса збіглася з надісланою. Це дозволяє підключати десятки різних сенсорів та модулів, використовуючи лише два контакти мікроконтролера.

Процес обміну даними регламентований наступною послідовністю етапів (рис. 5.6):

- 1) **Генерація стартової умови.** Main-пристрій ініціалізує сеанс зв'язку, перемикаючи лінію SDA з високого рівня на низький при збереженні стабільно високого рівня на лінії SCL. Це слугує сигналом для всіх Secondary-пристроїв про початок транзакції.
- 2) **Передача адреси.** Main по SDA транслює 7-бітну (інколи 10-бітну) адресу цільового пристрою. Усі Secondary-вузли порівнюють отриманий ідентифікатор із власним.
- 3) **Визначення режиму роботи (R/W біт).** Безпосередньо після адреси Main надсилає один біт, що визначає вектор руху даних. Значення «0» (Write) вказує на намір головного пристрою записати інформацію в периферійний вузол, а «1» (Read) – на запит щодо зчитування даних із нього.
- 4) **Підтвердження адреси (ACK).** Secondary-пристрій, чия адреса збіглася з транслованою, зобов'язаний підтвердити свою присутність на шині. Для цього він утримує SDA на низькому рівні (логічний нуль) протягом одного такту синхронізації, що генерується Main-ом.
- 5) **Цикл трансляції даних.** Обмін інформацією відбувається байтами (по 8 біт). Цей процес є ітераційним. У режимі запису Main передає дані, а Secondary приймає, а у режимі читання – навпаки. Після кожного байта отримувач (сторона, що приймала дані) надсилає біт підтвердження (ACK), сигналізуючи про успішне завершення прийому. Цикл повторюється доти, доки не буде передано весь масив даних або поки одна зі сторін не ініціює завершення обміну.

- 6) **Генерація умови зупинки.** Після завершення всіх операцій Main переводить лінію SDA з низького рівня у високий при високому стані лінії SCL. Ця дія завершує транзакцію та повертає шину в стан спокою, роблячи її доступною для наступних сеансів зв'язку.



Рис. 5.6. Структура кадру та послідовність передачі даних по шині I2C

Для керування шиною I2C з використанням фреймворку Arduino використовується бібліотека Wire [5]. Вона дозволяє абстрагуватися від низькорівневих процесів передачі сигналів, надаючи наступний набір готових методів:

- **wire.begin()** – ініціалізація I2C. При виклику без аргументу, пристрій отримує роль Main. При вказанні адреси в дужках – стає Secondary із відповідним ідентифікатором.

Запис даних (Main → Secondary):

- **wire.beginTransmission(адреса)** – контролер готує внутрішній буфер і запам'ятовує адресу пристрою, до якого спрямоване звернення.
- **wire.write(дані)** – виконується заповнення внутрішнього буфера корисним навантаженням. Дані не з'являються на лінії SDA, вони лише накопичуються в пам'яті контролера у черзі на відправку. Допускається послідовний виклик цього методу для формування багатобайтових пакетів.
- **wire.endTransmission()** – генерація стартової умови, трансляція адреси з бітом R/W = 0, передача вмісту буфера та завершення циклу умовою зупинки.

Читання даних (Main $\leftarrow$ Secondary):

- **Wire.requestFrom(*адреса*, *кількість*)** – надсилається запит до Secondary-пристрою на отримання певної кількості байтів. Метод є блокуючим – виконання програми продовжиться лише після отримання відповіді або виникнення помилки таймауту.
- **Wire.read()** – вилучає один байт із черги даних, що надійшли від Secondary-пристрою (після виклику команди запиту).

Обробка переривань для Secondary-пристроїв. Щоб другорядний пристрій міг асинхронно реагувати на команди, використовуються функції зворотного виклику (callback):

- **Wire.onReceive(*функція-обробник*)** – призначається функція-обробник, яка спрацьовує при отриманні пакетів даних від Main-пристрою.
- **Wire.onRequest(*функція-обробник*)** – призначається функція-обробник, яка викликається, коли Main-пристрій запитує інформацію.

Приклад:

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

void setup()
{
    // Ініціалізація I2C. Пристрій набуває ролі Main.
    Wire.begin();

    // Налаштування послідовного порту для моніторингу статусу
    Serial.begin(9600);
}

void loop()
{
```

```

// Підготовка до сесії з Secondary-пристроєм 0x27
Wire.beginTransmission(0x27);

// Наповнення буфера
Wire.write(0x01); // Номер регістру Secondary-пристрою
Wire.write(0xFF); // Значення, яке хочемо записати в нього

// Фізична трансляція даних на шину
byte status = Wire.endTransmission();

// Перевірка статусу передачі
if (status == 0)
{
    Serial.println("Дані успішно надіслано");
}
else
{
    Serial.print("Помилка передачі: ");
    Serial.println(status);
}

delay(1000); // Інтервал між транзакціями
}

```

MPU6050

Симулятор Wokwi забезпечує підтримку низки компонентів, що функціонують за протоколом I2C, серед яких MPU6050.

MPU6050 – це комбінований інерційний вимірювальний модуль (IMU), що поєднує трьохосьовий акселерометр, гіроскоп та датчик температури.

Взаємодія з модулем базується на зверненні до його внутрішніх регістрів. Кожна фізична величина представлена 16-бітним числом, розподіленим між двома

8-бітними регістрами (старший та молодший байти). Повна мапа регістрів та їх адресація наведені в документації [9].

Main-пристрій, для зчитування показників з підлеглого MPU6050, має виконати наступну послідовність дій:

- по I2C, разом з адресою MPU6050, надіслати байт, який MPU6050 сприйме як адресу внутрішньої комірки пам'яті, де зберігаються певні показники. При цьому слід заборонити генерацію Stop-умови, щоб зберегти контроль над шиною до наступного кроку;
- знову звернутись до MPU6050, але з активованим бітом читання (R/W біт = 1). Це ініціює перехід Main-пристрою в режим отримувача та встановлює кількість даних які він готовий прийняти, починаючи з раніше вказаної комірки пам'яті MPU6050;
- зчитувати дані по лінії SDA від MPU6050. Після кожного прийнятого байта Main-пристрій автоматично надсилає сигнал підтвердження (ACK), що стимулює MPU6050 перейти до наступного регістра та надіслати наступну порцію даних, аж поки Main-пристрій не отримає всі очікувані дані та не завершить комунікацію.

Отримані дані потребують подальшої програмної обробки, оскільки кожен показник (прискорення, кутова швидкість гіроскопа за кожною з осей та температура) представлений у модулі MPU6050 окремим 16-бітним числом. У внутрішній пам'яті пристрою ці значення розподілені між двома послідовними 8-бітними регістрами: один зберігає старшу частину числа (High Byte), а інший – молодшу (Low Byte). Через те, що інтерфейс I2C передає інформацію порціями лише по 8 біт, слід зчитувати послідовно два байти та об'єднувати їх у єдину змінну типу `int16_t` за допомогою побітового зсуву вліво на 8 розрядів та логічної операції «АБО» (`result = (High << 8) | Low`).

Після відновлення цілісності 16-бітних значень їх необхідно масштабувати для отримання реальних фізичних величин згідно з технічною документацією [9]. «Сирі» дані акселерометра діляться на коефіцієнт чутливості, який залежить від

обраного діапазону вимірювання (наприклад, 16384 для діапазону $\pm 2g$). Отриманий результат виражається в одиницях прискорення вільного падіння « g ». Для обчислення температури в градусах Цельсія отримане число (x) підставляється у формулу:

$$T = (x / 340,0) + 36,53.$$

Такий підхід забезпечує перетворення цифрових кодів, отриманих від MEMS-давачів, у зрозумілі для користувача одиниці вимірювання.

Завдання

1. Підключити червоний світлодіод через обмежувальний резистор до одного з вільних цифрових пінів Arduino UNO. Реалізувати керування даним світлодіодом за допомогою команд, отриманих через UART. При отриманні текстового рядка «**on**» світлодіод має вмикатися, а при «**off**» – вимикатися.
2. Підключити до мікроконтролера два зсувні регістри 74HC595. Використати спільні лінії даних (MOSI) і тактування (SCK) та окремі лінії (SS) для можливості вибору кожного з Secondary-пристроїв. До виходів кожного 74HC595 підключити по два RGB-світлодіоди. Реалізувати логіку, за якої команди на ввімкнення конкретного індикатора (**RGB1**, **RGB2**, **RGB3** та **RGB4**) надходять через UART, а безпосереднє керування регістрами для виконання цих команд здійснюється по протоколу SPI. При активації обраного світлодіода всі інші мають автоматично вимикатися.
3. Підключити до мікроконтролера два модулі MPU6050 за допомогою I2C. Для першого модуля використовувати адресу за замовчуванням (0x68), а для другого – встановити адресу **0x69**, підключивши його вивід AD0 до лінії живлення. Опитувати кожен з модулів щосекунди. З **0x68** знімати показники прискорення за усіма осями, а з **0x69** – показник температури. При цьому, забезпечити відображення показників у серійному моніторі.

Порядок виконання роботи

Крок 1. Виконати монтаж електронної схеми відповідно до представленої схеми з'єднань (рис. 5.7):

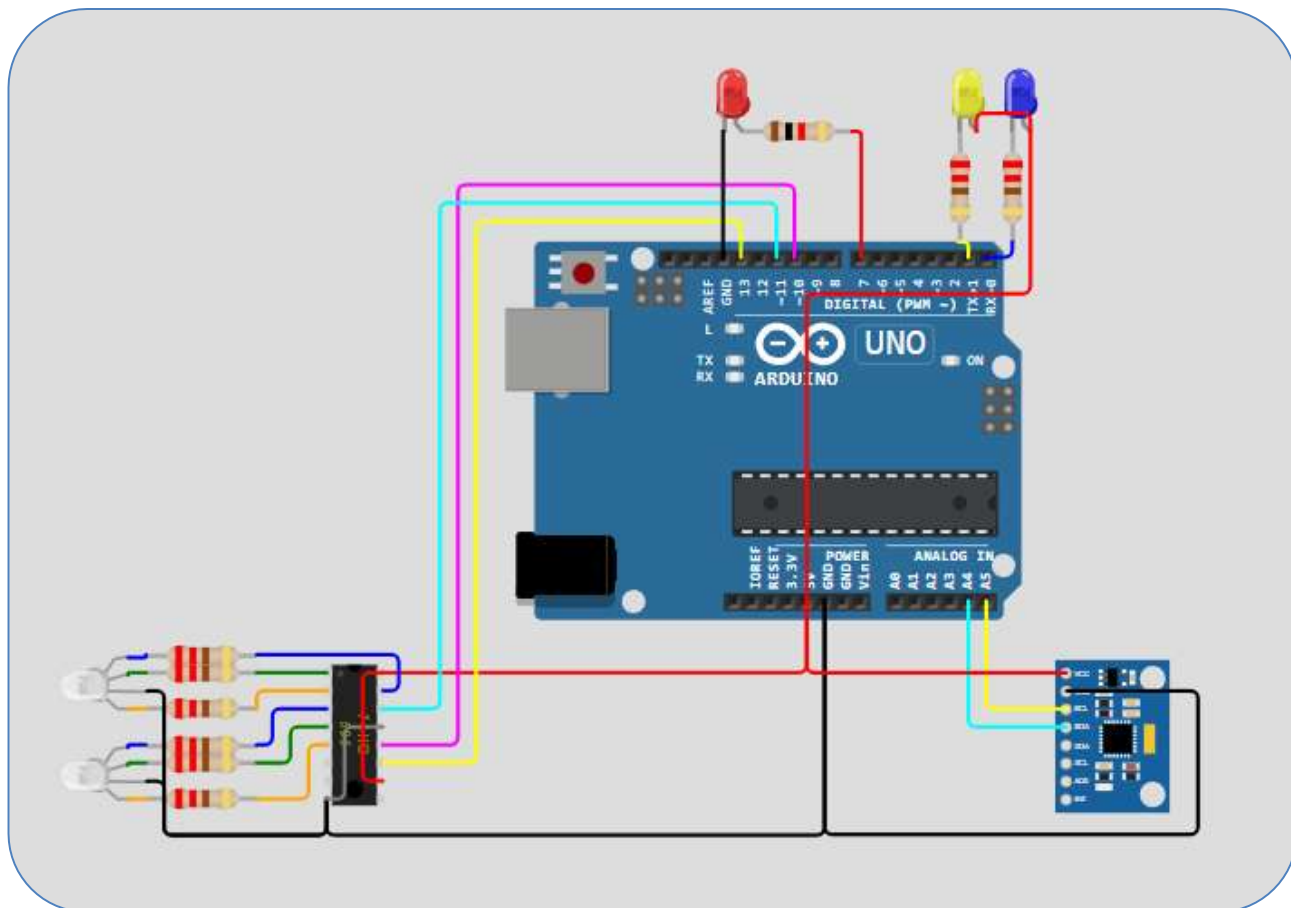


Рис. 5.7. Схема підключення периферійних пристроїв до мікроконтролера Arduino Uno

Крок 2. Здійснити компіляцію базового програмного коду та завантажити отриманий бінарний файл у флеш-пам'ять мікроконтролера Arduino UNO:

```
#include <Arduino.h>
#include <SPI.h>
#include <Wire.h>

#define REG_1_SS 10 // Визначення піна для вибору (SS) зсувного регістра
#define MPU_1_ADDRESS 0x68 // Визначення I2C-адреси першого сенсора MPU6050

// Зберігає час останнього виклику періодичних функцій (SPI та I2C)
unsigned long lastExecutionTime = 0;
```

```
//Оголошення прикладів функцій
void UART_example();
void SPI_example();
void I2C_example();

void setup()
{
    // Ініціалізація UART
    // Встановлення швидкості обміну даними з комп'ютером:
    Serial.begin(115200);
    // Виведення повідомлення у серійний монітор:
    Serial.println("Hello from UART!");

    //Ініціалізація SPI
    SPI.begin();
    pinMode(REG_1_SS, OUTPUT);

    //Ініціалізація I2C
    Wire.begin();
}

// Послідовний виклик кожного з прикладів в циклі loop
void loop()
{
    // UART перевіряється на кожній ітерації циклу (максимальна швидкість відгуку)
    UART_example();

    // Інші приклади виконуються рівно один раз на секунду, без блокування
    if (millis() - lastExecutionTime >= 1000)
    {
        lastExecutionTime = millis();
        SPI_example();
        I2C_example();
    }
}
```

```

    }
}

/***** UART *****/
void UART_example()
{
    if (Serial.available() > 0)
    {
        byte incomingByte = Serial.read();

        Serial.print(" Отримано: ");
        Serial.print(incomingByte);
        Serial.write(" Отримано: ");
        Serial.write(incomingByte);
        Serial.println("");
    }
}

/***** SPI *****/
void SPI_example()
{
    static byte ledState;

    if (ledState == 0)
    {
        // --- Увімкнення RGB1 з кольором Aqua ---
        digitalWrite(REG_1_SS, LOW);
        SPI.transfer(0b00000011);
        digitalWrite(REG_1_SS, HIGH);
        ledState = 1;
    }
    else
    {
        // --- Вимкнення RGB1 та RGB2 ---
        digitalWrite(REG_1_SS, LOW);

```

```

    SPI.transfer(0b00000000);
    digitalWrite(REG_1_SS, HIGH);
    ledState = 0;
}
}

/***** I2C *****/
void I2C_example()
{
    /* Дані акселерометра по кожній осі мають розмір 16 біт (2 байти).
       В пам'яті MPU6050 вони розділені на два 8-бітних регістри:
       H - High (старший) та L - Low (молодший).
       Можна зчитувати їх окремо, а потім об'єднувати. */

    // ЗЧИТУВАННЯ ПРИСКОРЕННЯ ПО ОСІ X
    // (покроковий метод, для демонстрації)

    // Підготовка до сесії з пристроєм за адресою MPU_1_ADDRESS:
    Wire.beginTransmission(MPU_1_ADDRESS);
    // Передача номеру регістру ACCEL_XOUT_H Secondary-пристрою:
    Wire.write(0x3B);
    // Фізична трансляція даних на шину, без генерації Stop-умови (false):
    Wire.endTransmission(false);
    // Запит на 1 байт до Secondary-пристрою, без генерації Stop-умови
    (false):
    Wire.requestFrom(MPU_1_ADDRESS, 1, false);
    // Читання та збереження першої (H) частини прискорення по осі X:
    byte xAxis_H = Wire.read();

    // Номер регістру ACCEL_XOUT_L Secondary-пристрою:
    Wire.write(0x3C);
    // Трансляція даних на шину, без генерації Stop-умови (false):
    Wire.endTransmission(false);
    // Запит на 1 байт до MPU та звільнення шини (true) після отримання:

```

```

Wire.requestFrom(MPU_1_ADDRESS, 1, true);
// Читання та збереження другої (L) частини прискорення по осі X:
byte xAxis_L = Wire.read();

// Об'єднання двох байтів у одне 16-бітне ціле число:
int16_t xAxis = word(xAxis_H, xAxis_L);

// Перетворення сирого значення у фізичні одиниці (g):
float xAxis_in_g = xAxis / 16384.0;

// --- Зчитування осей Y та Z разом (оптимізований метод) ---

// Підготовка до нової сесії з MPU_1:
Wire.beginTransmission(MPU_1_ADDRESS);
// Передача номеру регістру ACCEL_YOUT_H (старший байт осі Y):
Wire.write(0x3D);
// Фізична трансляція даних на шину, без генерації Stop-умови (false):
Wire.endTransmission(false);

// В пам'яті MPU6050 комірки Y та Z розташовані послідовно одна за одною
// Завдяки функції автоінкременту MPU6050, достатньо вказати лише
// стартову адресу (0x3D) та попросити пакет із 4 байтів поспіль
// (Y_H -> Y_L -> Z_H -> Z_L)

// Запит на 4 байти до MPU6050 та звільнення шини (true) після отримання:
Wire.requestFrom(MPU_1_ADDRESS, 4, true);
byte yAxis_H = Wire.read();
byte yAxis_L = Wire.read();
byte zAxis_H = Wire.read();
byte zAxis_L = Wire.read();

int16_t yAxis = word(yAxis_H, yAxis_L);
int16_t zAxis = word(zAxis_H, zAxis_L);

```

```

float yAxis_in_g = yAxis / 16384.0;
float zAxis_in_g = zAxis / 16384.0;

// --- Вивід отриманих даних ---
Serial.println("Прискорення: ");
Serial.println("По осі X = " + String(xAxis_in_g) + " g");
Serial.println("По осі Y = " + String(yAxis_in_g) + " g");
Serial.println("По осі Z = " + String(zAxis_in_g) + " g");
}

```

Крок 3. Шляхом моніторингу послідовного порту перевірити коректність функціонування та дослідити програмну реалізацію алгоритмів взаємодії пристроїв за інтерфейсами UART, SPI та I2C.

Крок 4. Програмне керування периферійними пристроями через UART (завдання №1):

- a. Визначити цифровий вихід мікроконтролера, до якого підключено червоний світлодіод, та задати для нього відповідний режим роботи (**OUTPUT**) у блоці початкових налаштувань.
- b. Модифікувати підпрограму **UART_example()**. Забезпечити повне зчитування вхідного текстового пакета (командного рядка) з приймального буфера UART одразу після фіксації наявності доступних для зчитування даних.
- c. У разі розпізнавання прийнятих команд «**on**» або «**off**» забезпечити встановлення відповідного (високого чи низького) логічного рівня на раніше налаштованому виході керування світлодіодом.

Крок 5. Масштабування шини SPI шляхом паралельного підключення зсувних регістрів (завдання №2):

- a. Інтегрувати до апаратної структури додаткову мікросхему зсувного регістра (74HC595). Інформаційні лінії (MOSI, SCK) підключити паралельно до існуючого інтерфейсу SPI. Лінію вибору пристрою (SS) другого чипа

вивести на окремий вільний цифровий пін мікроконтролера.

- b. Аналогічно наявному регістру – до вихідних портів другого зсувного регістру підключити два повноколірні RGB-світлодіоди (зі спільним катодом) через відповідні струмообмежувальні резистори.
- c. Модернізувати підпрограму **SPI_example()**, інтегрувавши в неї алгоритм розпізнавання та обробки додаткових текстових директив (**RGB1**, **RGB2**, **RGB3**, **RGB4**), що надходять каналом послідовного зв'язку UART.
- d. Залежно від ідентифікованої команди реалізувати алгоритм побайтового зсуву даних через SPI так, щоб відбувалася активація лише одного цільового RGB-компонента з одночасним переведенням решти світлодіодів у деактивований (вимкнений) стан.

Крок 6. Розширення адресації на шині I2C та зчитування температури (завдання №3):

- a. Підключити другий модуль MPU6050 до наявних фізичних ліній шини I2C (SDA та SCL).
- b. Задля запобігання адресному конфлікту змінити апаратну адресу другого модулю шляхом подачі VCC на його вихід AD0.
- c. Доповнити підпрограму **I2C_example()** алгоритмом послідовного опитування внутрішніх регістрів температури другого модуля та забезпечити циклічне виведення отриманих результатів у серійний монітор.

Порядок подання звіту

1. Виконайте вхід до курсу на платформі дистанційного навчання «Сікорський».
2. Перейдіть до розділу, що відповідає поточній роботі та відкрийте форму для подання результатів.
3. У полі для текстової відповіді додайте гіперпосилання на збережений проєкт у середовищі Wokwi.

Важливо! Перед відправленням переконайтеся у працездатності проекту запустивши симуляцію!

4. Збережіть відповідь для внесення проєкту до бази перевірки викладачем.
-

Контрольні запитання

1. Яка головна відмінність між асинхронним зв'язком (UART) та синхронним інтерфейсом (SPI)?
2. Навіщо в коді та в налаштуваннях монітора порту завжди виставляти однакову швидкість (Baud Rate)?
3. Скільки пристроїв можна з'єднати між собою за допомогою одного класичного порту UART?
4. Які три основні лінії (дроти) відповідають за передачу даних та тактування в протоколі SPI?
5. Яку функцію виконує пін SS під час роботи з мікросхемами по SPI?
6. Які два дроти використовуються для обміну даними та синхронізації в інтерфейсі I2C?
7. Як мікроконтролер розуміє, до якого саме давача на шині I2C він звертається в конкретний момент часу?
8. Який з трьох інтерфейсів (UART, SPI чи I2C) витрачає найменше пінів мікроконтролера для підключення великої кількості датчиків?

ПРАКТИЧНА РОБОТА № 6

ПІДКЛЮЧЕННЯ ДО БЕЗДРОТОВОЇ МЕРЕЖІ ТА ВИЗНАЧЕННЯ ТОЧНОГО МЕРЕЖЕВОГО ЧАСУ

Мета: Навчитися підключати мікроконтролерні пристрої до бездротових мереж Wi-Fi та виконувати синхронізацію системного часу за допомогою NTP-серверів.

План проведення практичного заняття

- Опрацювання теоретичних матеріалів.
 - Покрокове виконання завдань.
 - Подання результатів (звіту) на перевірку.
-

Теоретичні відомості

Технологія бездротового зв'язку Wi-Fi

Wi-Fi (Wireless Fidelity) – це технологія бездротової передачі даних, яка застосовується для об'єднання обладнання у локальні мережі (WLAN) або розгортання каналів зв'язку з глобальною мережею Інтернет. Ця технологія є одним із базових рішень для інтеграції програмованих засобів у єдину інформаційну систему, що дозволяє реалізувати дистанційний збір даних та віддалене адміністрування пристроїв.

Організація каналів зв'язку здійснюється виключно за допомогою радіохвиль, що виключає потребу в прокладанні дротових ліній (кабелів).

Робота обладнання відбувається у некомерційних смугах частот 2,4 ГГц (забезпечує більший радіус покриття та кращу проникність крізь перешкоди) та 5 ГГц (характеризується вищою пропускнуою здатністю та меншим рівнем завад).

Залежно від конкретного стандарту сімейства IEEE 802.11 (n, ac, ax), швидкість обміну даними може варіюватися від десятків мегабіт до кількох гігабіт за секунду.

Безпека інформаційних потоків і запобігання несанкціонованому доступу до мережі реалізуються за допомогою протоколів шифрування та автентифікації (напр. WPA2 чи WPA3).

Технологія Wi-Fi підтримується абсолютною більшістю сучасних обчислювальних засобів – від персональних комп'ютерів і смартфонів до спеціалізованих периферійних модулів Інтернету речей. Архітектура Wi-Fi дозволяє клієнтським пристроям автоматично та безперервно переключатися між різними точками доступу в межах однієї розподіленої мережі без втрати з'єднання.

Побудова бездротового вузла зв'язку може здійснюватися як на базі мікроконтролерів із вбудованим радіомодулем (наприклад, систем на кристалі сімейств ESP32 або ESP8266), так і шляхом підключення зовнішнього мережевого трансивера до основного пристрою через послідовні інтерфейси UART або SPI.

На програмному рівні взаємодія реалізується за допомогою спеціалізованих бібліотек (зокрема **wiFi.h**). Для успішної авторизації в коді програми обов'язково задаються ідентифікатор бездротової мережі (SSID) та відповідний ключ доступу (пароль).

Прошивка мікроконтролера повинна містити чітку послідовність дій для ініціалізації радіомодуля, циклічного моніторингу статусу поточного з'єднання та обробки позаштатних ситуацій (наприклад, автоматичного відновлення зв'язку в разі його обриву).

Приклад інтеграції мікроконтролера ESP32 у бездротову мережу за допомогою бібліотеки WiFi.h [5]:

- 1) На початку необхідно імпортувати базову бібліотеку для роботи з мережевим стеком:

```
#include <WiFi.h>
```

- 2) Задати текстові константи з ідентифікаторами Wi-Fi мережі:

```
const char* wifi_ssid = "Wokwi-GUEST"; //Назва мережі  
const char* wifi_pass = "";           //Пароль
```

Примітка. Для симулятора Wokwi віртуальна точка доступу розгортається з назвою мережі **Wokwi-GUEST** і не потребує введення пароля (порожній рядок).

3) Встановити один із підтримуваних режимів роботи Wi-Fi модуля:

- клієнтська станція – пристрій підключається до зовнішнього роутера або точки доступу:

```
WiFi.mode(WIFI_STA);
```

- точка доступу – мікроконтролер сам створює бездротову мережу для інших пристроїв:

```
WiFi.mode(WIFI_AP);
```

- комбінований режим – пристрій одночасно працює і як клієнт, і як точка доступу:

```
WiFi.mode(WIFI_AP_STA);
```

4) Запустити процес сканування та автентифікації передавши SSID та пароль від мережі у відповідний метод:

```
WiFi.begin(wifi_ssid, wifi_pass);
```

5) Для відстеження поточного стану мережевого підключення застосовують періодичне опитування модуля за допомогою методу **WiFi.status()**. Для підвищення читабельності коду програмісти використовують текстові назви статусів (так звані іменовані константи або макроси), насправді **WiFi.status()** повертає звичайне ціле число. Кожному цифровому коду відповідає конкретний стан мережевого стеку:

- **WL_IDLE_STATUS** (код 0) – базовий стан спокою. Модуль Wi-Fi ініціалізований, але ще не починав процедуру підключення;
- **WL_NO_SSID_AVAIL** (код 1) – помилка сканування ефіру. Радіомодуль не зміг виявити точку доступу з указаним ідентифікатором (SSID);
- **WL_SCAN_COMPLETED** (код 2) – проміжний статус. Мережевий модуль завершив сканування доступних бездротових мереж у зоні досяжності;

- **WL_CONNECTED** (код 3) – автентифікацію на точці доступу успішно завершено, пристрою виділено внутрішню IP-адресу. Система повністю готова до мережевого обміну;
- **WL_CONNECT_FAILED** (код 4) – спроба з'єднання відхилена точкою доступу (наприклад, через неправильно введений пароль);
- **WL_CONNECTION_LOST** (код 5) – аварійний стан, сигнал раптово зник або перервався вже після того, як стабільне з'єднання було встановлено;
- **WL_DISCONNECTED** (код 6) – модуль програмно відключено від мережі.

Оскільки процес підключення до мережі відбувається асинхронно і вимагає певного часу, програма не повинна переходити до виконання мережевих запитів одразу після виклику `WiFi.begin()`. Для блокування виконання програми до моменту отримання IP-адреси використовується циклічний моніторинг із затримкою:

```
// Цикл очікування успішного з'єднання
while (WiFi.status() != WL_CONNECTED)
{
    delay(500);           // Інтервал між повторними перевірками (0.5 секунд)
    Serial.print(".");    // Візуалізація процесу встановлення зв'язку в терміналі
}

```

Після успішного підключення можна зчитати призначену локальну IP-адресу та унікальний апаратний ідентифікатор (MAC-адресу) за допомогою методів `WiFi.localIP()` та `WiFi.macAddress()`:

```
Serial.print("Мережева адреса (IP): ");
Serial.println(WiFi.localIP());

Serial.print("Фізична адреса (MAC): ");
Serial.println(WiFi.macAddress());

```

Примітка. Документація з повним переліком класів та методів бібліотеки `WiFi.h`, а також прикладами доступна [\[5\]](#).

NTP (Network Time Protocol) та інструменти часової синхронізації

NTP – це протокол, призначений для синхронізації внутрішнього системного годинника обчислювальних пристроїв з розподіленими серверами мережевого часу. Використання NTP компенсує часовий дрейф, властивий автономним апаратним модулям реального часу (RTC), добова похибка яких може становити 20–30 секунд.

У вбудованих системах вибір методу обліку часу залежить від архітектурних вимог до автономності та точності пристрою. Мережева синхронізація є критично важливою для розподілених інфраструктур Інтернету речей (IoT), де маркування телеметричних даних (генерація часових міток) на різних пристроях має відбуватися за єдиною шкалою часу. Це забезпечує коректне логування подій у хмарних сховищах, синхронне виконання сценаріїв автоматизації («розумний будинок»), фіксацію хронології транзакцій та координацію мультимедійних потоків.

Для синхронізації часу за протоколом NTP може використовуватись бібліотека NTPClient [10]. Вона забезпечує обмін даними з серверами часу за транспортним протоколом UDP та надає набір готових методів для зчитування і налаштування часових параметрів мікроконтролера:

- **begin()** – виконує попереднє налаштування об'єкта NTP-клієнта. Даний метод викликають один раз у блоці **setup**;
- **update()** – надсилає запит до сервера та оновлює локальні значення часу;
- **getFormattedTime()** – повертає поточний час у вигляді текстового рядка (HH:MM:SS);
- **getEpochTime()** – повертає кількість секунд, що минули з 1 січня 1970 року;
- **setTimeOffset(зміщення)** – приймає ціле число у секундах, яке додається до отриманого від сервера часу UTC. Оскільки NTP-сервери завжди транслюють всесвітній час UTC 0, для відображення коректного часу в Україні цей метод викликають із параметром **7200** (для зимового часу, UTC+2) або **10800** (для літнього часу, UTC+3);

- **setUpdateInterval(*інтервал*)** – визначає, як часто метод **update()** буде надсилати реальні запити в Інтернет, що захищає мережу від перевантаження. Якщо встановити інтервал, наприклад, 60000 мілісекунд (1 хвилина), то навіть при щосекундному виклику **update()** у циклі **loop()**, запити до сервера відправлятимуться лише раз на хвилину;
- **getHours()**, **getMinutes()**, **getSeconds()** – розраховують і повертають окремі числові компоненти поточного часу;
- **getDay()** – повертає цифровий код поточного дня тижня (число від 0 до 6). Оскільки відлік ведеться за американським стандартом, в неділю повертається 0, в понеділок – 1, а в суботу – 6.

Приклад отримання точного мережевого часу на мікроконтролері ESP32 за допомогою бібліотеки NTPClient [\[10\]](#):

- 1) У початковій частині програми необхідно підключити бібліотеку для роботи з протоколом NTP, а також допоміжну бібліотеку, яка забезпечує транспортування пакетів по UDP:

```
#include <Arduino.h>
#include <NTPClient.h>
#include <WiFiUdp.h>
```

- 2) У глобальній області видимості необхідно оголосити екземпляри класів для керування мережевими протоколами – об'єкт типу **WiFiUDP** для організації транспортного рівня та об'єкт **NTPClient**, у конструктор якого слід передати посилання на створений UDP-транспорт, адресу сервера часу і величину годинного зсуву відносно UTC у секундах:

```
WiFiUDP udp_transport;
NTPClient ntp_client(udp_transport, "pool.ntp.org", 7200);
```

- 3) У межах функції первинного налаштування **setup()**, суворо після успішного встановлення Wi-Fi з'єднання, потрібно викликати метод

ntp_client.begin(), який відкриває локальний UDP-порт і переводить клієнт у стан готовності до надсилання мережових запитів:

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

    // Код підключення до Wi-Fi
    ntp_client.begin();
}
```

- 4) У головному циклі програми **loop()** необхідно періодично викликати метод **update()**, який за потреби надсилає запит до сервера та коригує внутрішній лічильник часу:

```
void loop()
{
    ntp_client.update();

    Serial.print("Час: ");
    Serial.println(ntp_client.getFormattedTime());

    delay(1000);
}
```

Коли викликається метод **ntp_client.update()**, бібліотека отримує від сервера значення часу і записує його у внутрішню змінну, яка є частиною створеного об'єкта **ntp_client**. Коли викликаються методи **getFormattedTime()** або **getHours()**, бібліотека бере збережене в оперативній пам'яті значення часу, додає до нього кількість мілісекунд, що минули з моменту останньої синхронізації, і видає актуальний результат. Оскільки оперативна пам'ять є енергозалежною, у разі перезавантаження або вимкнення живлення мікроконтролера ці дані повністю втрачаються. При кожному новому старті пристрою процедуру підключення до Wi-Fi та синхронізацію необхідно виконувати заново.

Виведення даних на рідкокристалічний дисплей

Для візуалізації текстової інформації в вбудованих системах часто використовуються рідкокристалічні дисплеї. Широко розповсюдженими є дисплеї LCD 1602 та LCD 2004, особливо їх I2C-версії. Задля економії виводів мікроконтролера їх підключають через двопровідну послідовну шину I2C.

Для спрощення програмного керування даними дисплеями існує бібліотека `LiquidCrystal_I2C.h` [11], яка містить високорівневі методи для налаштування та виведення даних. Після її підключення, слід обов'язково оголосити екземпляр класу, передавши у конструктор I2C-адресу потрібного дисплея та параметри його матриці:

```
#include <LiquidCrystal_I2C.h>

// Оголошення об'єкта для дисплея на 20 стовпців та 4 рядки
LiquidCrystal_I2C lcd(0x27, 20, 4);
```

Базові методи бібліотеки `LiquidCrystal_I2C` [11]:

- **init()** – виконує первинне налаштування (у блоці **setup**) внутрішніх регістрів контролера екрана, конфігурує режим роботи;
 - **setCursor(стовпець, рядок)** – встановлює позицію невидимого покажчика (курсора) у зазначений стовпець та рядок. Відлік координат починається з нуля (лівий верхній кут екрана має координати 0, 0);
 - **print(текст)** – виводить текстовий рядок або окремі символи на екран автоматично зсуваючи курсор праворуч;
 - **home()** – повертає курсор у початкову позицію (0, 0) без зміни або видалення раніше виведеної на екран інформації;
 - **clear()** – видаляє всі символи з екрана, після чого переміщує курсор у початкову позицію (0, 0).
-

Завдання

1. Запрограмувати виведення поточного стану Wi-Fi на рідкокристалічний дисплей. До моменту авторизації та після відключення мережі – відображати текст «**Offline**», під час підключення – «**Connecting**», а при встановленні з'єднання – «**Online**» та призначену ESP32 IP-адресу.
2. Налаштувати отримання часових міток від пулу NTP-серверів. Забезпечити коректне відображення годин, хвилин та секунд на дисплеї з обов'язковим врахуванням регіонального часового зсуву.
3. Реалізувати комбінований алгоритм ходу годинника. Пряме звернення до NTP-сервера має відбуватися виключно раз на хвилину. Протягом 60-секундного інтервалу очікування кожна секунда на екрані повинна інкрементуватися автономно, спираючись на внутрішній системний час мікроконтролера.

Додаткове завдання 1. Реалізувати можливість примусового від'єднання від мережі Wi-Fi (напр. за допомогою тактової кнопки). Хід годинника повинен безперервно тривати в автономному режимі, спираючись на внутрішній таймер мікроконтролера.

Додаткове завдання 2. Розширити схему двома тактовими кнопками для можливості коригування часу. Натискання першої кнопки має миттєво зміщувати поточний час на одну годину вперед (+1 год), а другої – на одну годину назад (-1 год).

Важливо! Бали за додаткові завдання нараховуються за умови вчасного подання звіту.

Порядок виконання роботи

Крок 1. Керуючись схемою на рис. 6.1, виконати фізичне підключення дисплея LCD 2004 до відповідних GPIO-виводів мікроконтролера ESP32.

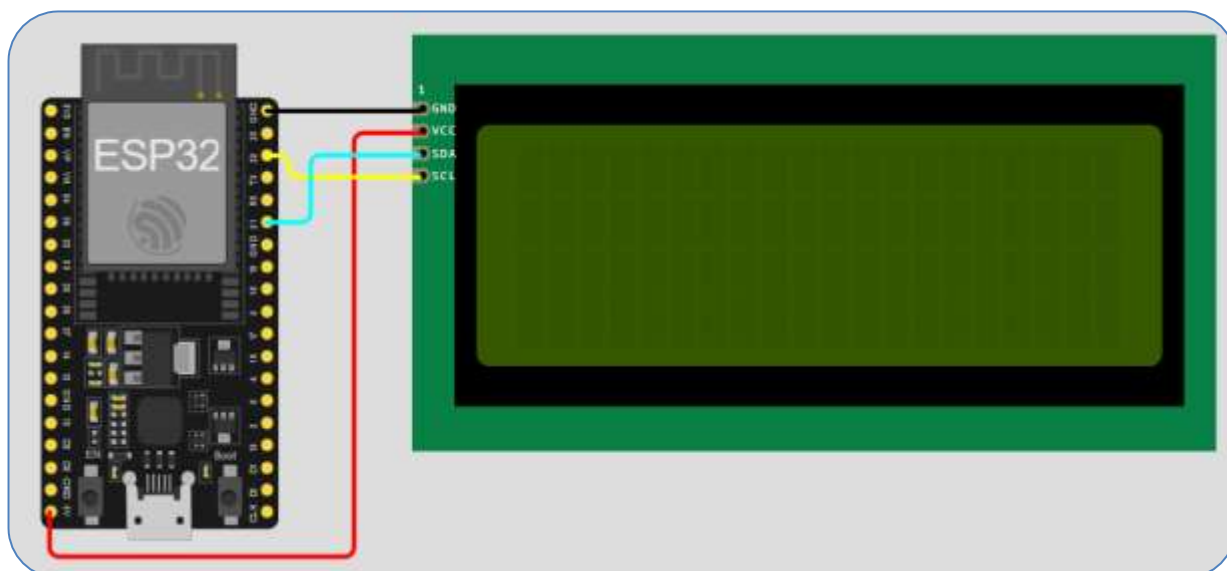


Рис. 6.1. Схема з'єднання дисплея з мікроконтролером

Крок 2. У кодї програми оголосити об'єкт дисплея, задавши його розміри (кількість стовпчиків і рядків) та шістнадцяткову адресу.

Крок 3. Використовуючи методи **Wi-Fi.h**, реалізувати алгоритм підключення мікроконтролера до локальної точки доступу.

Крок 4. Запрограмувати визначення мережевого статусу. До моменту з'єднання виводити на дисплей повідомлення про автономну роботу (**Offline**), а після успішного підключення – змінювати текст на «**Online**».

Крок 5. Визначити локальну IP-адресу, видану DHCP-сервером маршрутизатора, та відобразити її на дисплеї.

Крок 6. Створити та налаштувати екземпляр клієнта для взаємодії з розподіленою мережею серверів часу. Впровадити часовий зсув у секундах, що відповідає поточному поясу України.

Крок 7. Обмежити прямі звернення до NTP-сервера так, щоб вони відбувалися суворо один раз в 60 секунд.

Крок 8. Налаштувати безперервне щосекундне виведення поточного часу на I2C-дисплей у форматі **HH:MM:SS**.

Порядок подання звіту

1. Виконайте вхід до курсу на платформі дистанційного навчання «Сікорський».
2. Перейдіть до розділу, що відповідає поточній роботі та відкрийте форму для подання результатів.
3. У полі для текстової відповіді додайте гіперпосилання на збережений проєкт у середовищі Wokwi.

Важливо! Перед відправленням переконайтеся у працездатності проєкту запустивши симуляцію!

4. Збережіть відповідь для внесення проєкту до бази перевірки викладачем.
-

Контрольні запитання

1. Що таке мережеві ідентифікатори?
2. Який метод використовується для моніторингу поточного стану мережевого з'єднання?
3. Які існують стани (статуси) Wi-Fi з'єднання у бібліотеці WiFi.h?
4. Для чого призначений протокол мережевого часу (NTP)?
5. В чому різниця між координованим всесвітнім часом (UTC) та Unix-епохою?
6. Яким чином відбувається відлік часу між запитами до NTP-сервера?

ПРАКТИЧНА РОБОТА № 7

КЛІЄНТ-СЕРВЕРНА КОМУНІКАЦІЯ ЗА ПРОТОКОЛОМ HTTP

Мета: Навчитися організовувати обмін даними між мікроконтролером і веб-сервером за протоколом HTTP, освоїти принципи клієнт-серверної взаємодії та практично застосувати їх для отримання і відправки даних.

План проведення практичного заняття

- Опрацювання теоретичних матеріалів.
 - Покрокове виконання завдань.
 - Подання результатів (звіту) на перевірку.
-

Теоретичні відомості

Мережева взаємодія на прикладному рівні

В основі інформаційного обміну сучасного Інтернету лежить протокол **HTTP (Hypertext Transfer Protocol)**. Він стандартизує правила, за якими клієнтські застосунки (від стандартних веббраузерів до автономних IoT-модулів) взаємодіють із віддаленими серверами для отримання або надсилання медіаконтенту, структурованих документів (HTML), сирих даних тощо.

Робота протоколу базується на трьох простих принципах:

- 1) **Взаємодія «запит-відповідь».** Організатором зв'язку завжди виступає клієнт. Браузер або IoT-пристрій надсилає запит, а сервер лише реагує на нього – повертаючи потрібні дані або повідомлення про помилку.
- 2) **Незалежність запитів (Stateless).** Сервер не запам'ятовує історію спілкування з клієнтом. Кожне нове звернення для нього виглядає так, ніби воно перше. Щоб сайт «впізнавав» користувача (наприклад, тримав його в особистому кабінеті), розробники використовують додаткові інструменти, як-от файли cookies або спеціальні токени.

3) **Службові заголовки.** Кожне повідомлення в мережі містить не лише корисний вміст, а й приховані технічні рядки (заголовки). Вони підказують пристроям, що саме передається (наприклад, текст чи картинка) та як ці дані обробляти.

Коли пристрій звертається до сервера, він обов'язково додає до запиту конкретну команду (метод), яка вказує на бажану дію. Найчастіше використовуються такі методи:

- **GET** – прохання просто показати або завантажити дані.
- **POST** – відправка нової інформації на сервер (наприклад, тексту з форми реєстрації).
- **PUT** та **PATCH** – команди для оновлення чи виправлення інформації, яка вже є на сервері.
- **DELETE** – запит на видалення певного файлу чи сторінки.
- **HEAD** – завантаження лише технічних заголовків без самого вмісту сторінки, що допомагає економити інтернет-трафік.

Отримавши команду, сервер обов'язково повертає цифровий код, який показує результат роботи. Наприклад, код **200 OK** означає, що все пройшло успішно і дані доставлено. Якщо сторінку назавжди перенесли на іншу адресу, сервер видасть код **301 Moved Permanently**. Помилка **404 Not Found** свідчить про те, що користувач ввів неправильну адресу і такої сторінки не існує. А код **500 Internal Server Error** вказує на внутрішній збій у програмі самого сервера.

Слід враховувати, що протокол HTTP має суттєвий недолік – всі дані передаються у відкритому вигляді. Будь-який зловмисник у мережі (наприклад, у публічній Wi-Fi зоні) може перехопити паролі чи номери банківських карток тощо. Щоб цього не сталося, створили захищену версію протоколу – HTTPS («S» – Secure). Його відмінність полягає в тому, що стандартні HTTP-повідомлення шифруються криптографічним протоколом TLS.

Незважаючи на розвиток мережових технологій і появу нових версій протоколу, більшість сучасних сайтів та сервісів досі використовують класичну версію HTTP/1.1.

Розглянемо приклад передачі телеметрії від датчика до сервера. Після виконання вимірювань (температура та заряд батареї) пристрій формує HTTP-запит. Для додавання нових записів до бази даних сервера оптимальним є використання методу POST:

```
POST /api/telemetry.php HTTP/1.1
Host: gateway.example.net
Content-Type: text/plain; charset=utf-8
Content-Length: 38

dev_id=ESP32_01, temp=36, battery=3.72
```

Запит складається з:

- Стартового рядка, що містить метод запиту (**POST**), шлях до скрипта-обробника (**telemetry.php**) на стороні сервера та версію протоколу (HTTP/1.1).
- Заголовків, що передають метадані про адресу призначення, формат кодування інформації та точний обсяг повідомлення в байтах.
- Тіла запиту, що містить головне корисне навантаження – ID пристрою, а також виміряні параметри (температура та напруга акумулятора) у вигляді звичайного тексту.

Отримавши цей пакет, сервер базової станції передає його вміст на обробку скрипту **telemetry.php**. Серверна програма (частина присвячена обробці **POST**-запитів) зчитує корисне навантаження з тіла запиту, розпізнає ідентифікатор пристрою та заносить отримані телеметричні показники до бази даних. Після успішного збереження інформації сервер обов'язково формує та надсилає відповідь, яка підтверджує результат операції:

```
HTTP/1.1 200 OK
```

```
Content-Type: text/plain; charset=utf-8
```

```
Content-Length: 23
```

```
Data successfully saved
```

HTTP-відповідь має чітку структуру та включає:

- Рядок статусу, що починається з версії протоколу, за якою йде трицифровий код стану та його текстове розшифрування. Код 200 повідомляє, що запит оброблено успішно, без збоїв на стороні сервера. Якщо замість коду 200 повернеться інший статус (наприклад, 500 або 404), слід запустити алгоритм повторної відправки або зберегти поточну телеметрію у внутрішню пам'ять, щоб надіслати її пізніше, коли сервер відновить роботу.
- Заголовки, що передають службові метадані, наприклад **Content-Length**. Аналізуючи **Content-Length**, пристрій порівнює вказану сервером кількість байт із фактично отриманим обсягом даних у буфері. Якщо значення не збігаються (наприклад, через втрату пакетів), мікроконтролер ініціює скидання поточного з'єднання як нестабільного.
- Тіло, у вигляді тексту, який мікроконтролер може зчитати та вивести у термінал для підтвердження того, що телеметрія дійсно дійшла до бази даних. Крім цього, тіло відповіді може містити керуючі директиви для пристрою – наприклад, команди на зміну періодичності замірів, дистанційне перезавантаження або переведення системи в енергозберігаючий режим.

Головними перевагами HTTP є його глобальна стандартизація, простота інтеграції в програмне забезпечення мікроконтролерів та сумісність із будь-якою сучасною веб-інфраструктурою. Проте в контексті автономних радіоелектронних пристроїв виявляються суттєві архітектурні недоліки, оскільки радіопередавач змушений витрачати енергію на транспортування великих обсягів службового тексту (супровідні заголовки HTTP часто в рази перевищують обсяг самих телеметричних даних). У таких умовах значно ефективнішим рішенням є використання спеціалізованих протоколів (наприклад, MQTT), які оптимізовані

для передачі малих пакетів даних, мають мінімальні заголовки та підтримують контроль якості доставки повідомлень (QoS).

Форматування та обробка даних

Серіалізація – це процедура трансформації внутрішніх структур даних програми (змінних, масивів або об'єктів) у лінійний послідовний формат (наприклад, текстовий рядок), придатний для запису в енергонезалежну пам'ять або відправки через канали зв'язку.

Десеріалізація – зворотна процедура, під час якої отриманий ззовні байтовий або текстовий потік розбирається (парситься) для відновлення вихідних значень і запису їх у змінні програми.

Для ілюстрації проблеми розглянемо такий фрагмент коду з набором даних:

```
int id = 14;
double temperatura = 18.4;
double vologist = 35;
String misto = "Київ";
double koordynaty [] = {50.44754, 30.45823};
```

Якщо об'єднати ці значення в суцільний неструктурований рядок вигляду «1418.435Київ50.4475430.45823», приймальній стороні буде важко коректно розділити суміжні числові параметри. Для розв'язання цієї проблеми застосовують стандартизовані текстові формати, найпопулярнішим серед яких є **JSON (JavaScript Object Notation)** [\[12\]](#).

У форматі JSON аналогічний набір даних набуває чіткої структури «ключ – значення»:

```
{
  "id": 14,
  "temperatura": 18.4,
  "vologist": 35,
  "misto": "Київ",
```

```
"koordynaty": [50.44754, 30.45823]
}
```

Для десеріалізації такого JSON-тексту доцільно використовувати бібліотеку `ArduinoJson` [13]. Підключивши бібліотеку, необхідно визначити джерело вхідних даних. У наведеному нижче прикладі дані задано як змінну типу **String**, хоча в реальних сценаріях текстовий рядок зазвичай надходить через послідовний порт або бездротовий інтерфейс. Для того, щоб компілятор не сприйняв внутрішні лапки як кінець текстового блоку, застосовується символ екранування (зворотний слеш «\»). Далі створюється універсальний об'єкт-контейнер **JsonDocument**, призначений для збереження та зручного доступу до розібраної JSON-структури:

```
#include <ArduinoJson.h>

// Текстового рядка з даними у форматі JSON
String data =
"{\"id\":14,\"temperatura\":18.4,\"vologist\":35,\"misto\":\"Київ\",
\"koordynaty\":[50.44754,30.45823]}";

// Створення об'єкта-контейнера для зберігання структури даних
JsonDocument json;
```

Функція **deserializeJson** приймає створений контейнер та вхідний рядок, після чого виконує посимвольний аналіз тексту. Алгоритм автоматично верифікує правильність розставлених дужок, двокрапок та ком. Якщо структура валідна, функція заповнює об'єкт **json** зв'язками між ключами та їхніми значеннями:

```
// Виклик функції розбору даних та заповнення об'єкта json
deserializeJson(json, data);
```

Після успішного наповнення об'єкта **json** стає доступним пряме звернення до його елементів за їхніми текстовими іменами (ключами). Важливо враховувати, що парсер чутливий до регістру літер. Бібліотека `ArduinoJson` автоматично виконує

конвертацію типів під час присвоєння. Достатньо оголосити цільову змінну відповідного типу, і дані з JSON-структури будуть зведені до **int**, **double** чи **String**:

```
// Вилучення значень із автоматичною конвертацією типів
int id          = json["id"];
double temperatura = json["temperatura"];
double vologist  = json["vologist"];
String misto     = json["misto"];
```

Окремої уваги вимагає ключ «**koordynaty**», оскільки він містить не поодинокі значення, а масив із двох чисел. Для доступу до таких даних використовується індексація, де числа **0** та **1** визначають порядковий номер елемента в масиві. Отримані значення широти та довготи записуються у дві локальні змінні, після чого програма об'єднує їх у стандартний масив мови C++ для зручності подальшого оброблення або виведення:

```
// Поелементне вилучення координат за цифровими індексами масиву
double Shyrota = json["koordynaty"][0]; // Перший елемент (широта)
double Dovhota = json["koordynaty"][1]; // Другий елемент (довгота)

// Локальне збереження координат у класичний масив C++
double koordynaty[] = {Shyrota, Dovhota};
```

Описаний підхід є безпечним методом обробки структурованої інформації, проте у випадках, коли обсяг оперативної пам'яті мікроконтролера сильно обмежений або вхідні дані мають просту структуру, використання важких бібліотек типу **ArduinoJson** може бути недоцільним. Альтернативним варіантом є застосування вбудованого інструментарію класу **String**, який дозволяє виконувати сегментування тексту, пошук маркерів та виокремлення числових параметрів за допомогою індексів символів.

Для того щоб вирізати корисні дані з суцільного рядка, програма повинна знайти координати символу-роздільника (наприклад, двокрапки або коми). Метод **indexOf(символ)** сканує текст зліва направо і повертає порядковий номер (індекс)

першого входження вказаного знака. Якщо такий маркер у рядку відсутній, метод повертає **-1**.

```
String data = "temp:18.4";  
  
// Визначення позиції розділювального знака в текстовому масиві  
int index = data.indexOf(':'); // Результат виконання: індекс 4
```

Після визначення точних координат роздільника використовується метод **substring(початковий_індекс, кінцевий_індекс)**. Він копіює та повертає фрагмент тексту, розташований між вказаними позиціями, при цьому символ на кінцевій позиції до нового підрядка не включається. Якщо передати методу лише один параметр (початковий індекс), обрізання відбудеться від цієї точки і до самого кінця рядка:

```
String cleanValue = data.substring(index + 1); // Результат: "18.4"
```

Завдання

1. Налаштувати мікроконтролер для виконання мережевого запиту (**HTTPS GET**) до веб-ресурсу Українського гідрометеорологічного центру за адресою: <https://www.meteo.gov.ua/ua/Pohoda-na-sohodni>. Отриману від сервера відповідь необхідно відобразити у серійному моніторі.
2. Реалізувати взаємодію з відкритим сервісом надання погодних даних, надіславши запит до інтерфейсу програмування додатків (API): https://api.open-meteo.com/v1/forecast?latitude=50.44754&longitude=30.45823&hourly=temperature_2m&timezone=auto&forecast_days=1.

Примітка. Ознайомитися з повною специфікацією API та виконати індивідуальне конфігурування параметрів запиту (обрати інші географічні координати, змінити часовий пояс або додати додаткові метеорологічні метрики) можна за допомогою інтерактивного конструктора за посиланням [14].

3. З отриманого масиву даних у форматі JSON виокремити показник температури навколишнього середовища, що відповідає часу за варіантом (див. табл. 7.1). Оброблений результат структурувати та вивести на рідкокристалічний дисплей у такому вигляді: «T(14:00) = 10.5 *C».
4. Створити унікальну URL-адресу для тестування за допомогою сайту <https://webhook.site/>. Кожні 15 секунд визначати температуру оточуючого середовища (за допомогою DHT22) та відправляти її значення на згенеровану адресу за протоколом HTTP.

Додаткове завдання 1. За допомогою базових методів обробки рядків виокремити числовий показник температури з відповідь від сервера (**payload**), отриманої під час виконання першого завдання, та вивести його на екран (напр. «В Києві: -1 °C»).

Додаткове завдання 2. Додати ще один дисплей 20x4 та, використовуючи масив даних з другого завдання, налаштувати виведення добового прогнозу погоди у вигляді чотирьох послідовних часових зрізів: ранкового, денного, вечірнього та нічного.

Важливо! Бали за додаткові завдання нараховуються за умови вчасного подання звіту.

Таблиця варіантів

Таблиця 7.1. Часові мітки прогнозу погоди для індивідуальних варіантів

Вар.	0	1	2	3	4	5	6	7	8	9
0	—	01:00	02:00	03:00	04:00	05:00	06:00	07:00	08:00	09:00
10	10:00	11:00	12:00	13:00	14:00	15:00	16:00	17:00	18:00	19:00
20	20:00	21:00	22:00	23:00	00:00	01:00	02:00	03:00	04:00	05:00
30	06:00	07:00	08:00	09:00	10:00	11:00	12:00	13:00	14:00	15:00
40	16:00	17:00	18:00	19:00	20:00	21:00	22:00	23:00	00:00	01:00
50	02:00	03:00	04:00	05:00	02:00	03:00	04:00	05:00	06:00	07:00

Порядок виконання роботи

Крок 1. Керуючись схемою (рис. 7.1), виконати підключення периферійних модулів до мікроконтролера ESP32:

- а. рідкокристалічний дисплей 20x4 під'єднати до шини I2C (лінії SDA та SCL);
- б. цифровий датчик температури та вологості DHT22 підключити до визначеного сигнального порту GPIO;
- с. світлодіод через обмежувальний резистор підключити до відповідного цифрового виходу.

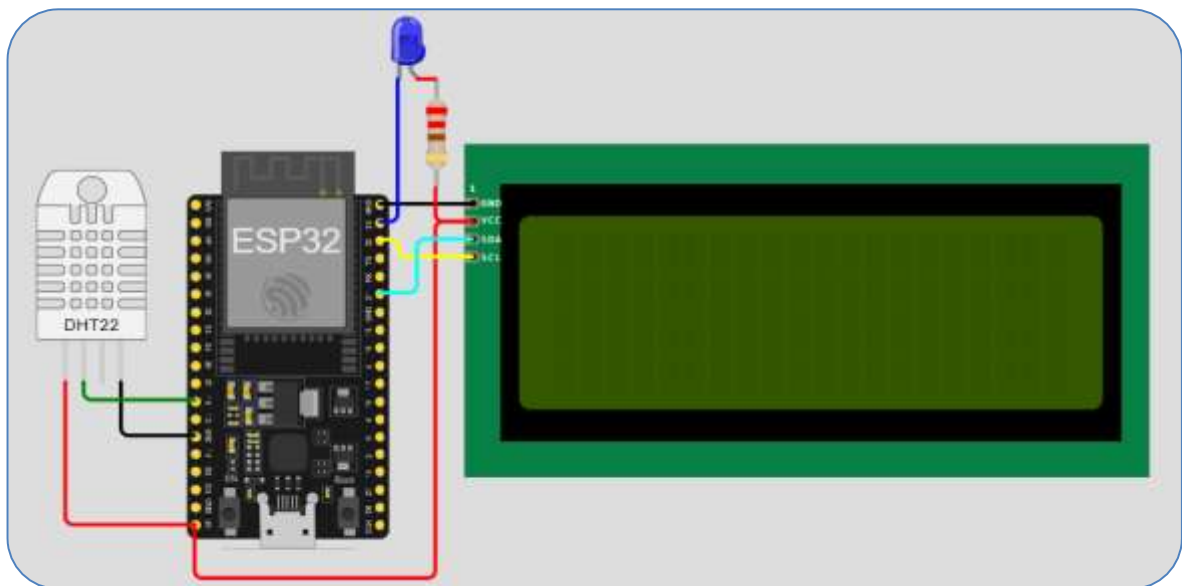


Рис. 7.1. Схема підключення датчика DHT22, світлодіода та LCD-дисплея до ESP32

Крок 2. Підключити бібліотеки для роботи з периферією та мережею (`LiquidCrystal_I2C.h`, `WiFi.h`, `HTTPClient.h`). Задати параметри точки доступу, адресу цільового сервера та ініціалізувати об'єкт дисплея. У функції `setup()` реалізувати алгоритм підключення до бездротової мережі з виводом на дисплей статусу.

Крок 3. Додати та дослідити функцію для взаємодії з віддаленим сервером «<https://www.meteo.gov.ua/ua/Pohoda-na-sohodni>», яка має перевіряти наявність підключення, формувати HTTP-запит, зчитувати код відповіді та повертати

корисне навантаження (**payload**) у серійний монітор:

```
const String url1 = "https://www.meteo.gov.ua/ua/Pohoda-na-sohodni";

void fetchWeatherHTML()
{
    // Верифікація активності мережевого інтерфейсу
    if (WiFi.status() != WL_CONNECTED)
    {
        Serial.println("Помилка: відсутній зв'язок із мережею");
        return;
    }

    HTTPClient http;

    // Ініціалізація безпечного з'єднання URL
    if (http.begin(url1))
    {
        int responseCode = http.GET(); // Виконання GET-запиту

        // Обробка результату відповіді сервера
        if (responseCode > 0)
        {
            Serial.printf("HTTP Статус: %d\n", responseCode);
            // Зчитування та виведення вмісту сторінки
            String payload = http.getString();
            Serial.println(payload);
        }
        else
        {
            Serial.print("Невдалий запит. Код помилки: ");
            Serial.println(responseCode);
        }
    }
}
```

```
http.end(); // Звільнення ресурсів підключення
}
}
```

Крок 3. Модернізувати або продублювати функцію відправки запиту для роботи з наступним сервером (https://api.open-meteo.com/v1/forecast?latitude=50.44754&longitude=30.45823&hourly=temperature_2m&timezone=auto&forecast_days=1).

Забезпечити виведення у серійний монітор сирого тексту відповіді (у форматі JSON), що містить погодинний прогноз.

Крок 4. Використовуючи інструментарій бібліотеки ArduinoJson, реалізувати синтаксичний розбір масиву даних:

- створіть об'єкт **JsonDocument** та наповніть його даними отриманими на попередньому кроці;
- здійсніть вибірку показника, що відповідає індексу години індивідуального варіанту (див. табл. 7.1);
- додайте на дисплей інформацію за шаблоном: «Т(14:00) = 10.5 *С».

Крок 5. Визначити локальну температуру за допомогою датчика DHT22. Отримане значення упакувати в HTTP-запит (наприклад, у вигляді GET-аргументів або JSON-payload) і надіслати на сервер за адресою згенерованою за допомогою <https://webhook.site/>.

Порядок подання звіту

- Виконайте вхід до курсу на платформі дистанційного навчання «Сікорський».
- Перейдіть до розділу, що відповідає поточній роботі та відкрийте форму для подання результатів.
- У полі для текстової відповіді додайте гіперпосилання на збережений проєкт у середовищі Wokwi.

Важливо! Перед відправленням переконайтеся у працездатності проекту запусивши симуляцію!

4. Збережіть відповідь для внесення проєкту до бази перевірки викладачем.
-

Контрольні запитання

1. Яку структуру має HTTP-запит і які обов'язкові компоненти входять до його складу?
2. Чому використання стандартного текстового протоколу HTTP інколи вважається надмірним?
3. Для чого у програмі аналізується числовий код HTTP-відповіді?
4. У чому полягає фізична сутність процесів серіалізації та десеріалізації даних під час міжмережевого обміну?
5. Які структурні особливості формату JSON?
6. Як реалізується доступ до елементів вкладеного масиву під час роботи з об'єктом **JsonDocument** бібліотеки **ArduinoJson**?
7. Які наслідки для логіки роботи програми матиме помилка в реєстрі символів (наприклад, **id** замість **Id**) при зверненні до ключа в бібліотеці **ArduinoJson**?
8. Для виконання яких операцій призначені вбудовані методи **indexOf** та **substring** класу **String**?

ПРАКТИЧНА РОБОТА № 8

ЗАСТОСУВАННЯ ОПЕРАЦІЙНОЇ СИСТЕМИ РЕАЛЬНОГО ЧАСУ (RTOS)

Мета: Набуття практичних навичок розподілу ресурсів процесора між окремими потоками виконання (завданнями) за допомогою операційної системи реального часу.

План проведення практичного заняття

- Опрацювання теоретичних матеріалів.
 - Покрокове виконання завдань.
 - Подання результатів (звіту) на перевірку.
-

Теоретичні відомості

Операційна система (ОС) – це ПЗ, що забезпечує взаємодію між апаратними ресурсами мікроконтролера та прикладними програмами. Її головна мета полягає в ефективному управлінні процесорним часом, оперативною пам'яттю та периферійними інтерфейсами.

Інтеграція ОС у вбудовані системи істотно спрощує процес розробки завдяки механізмам абстракції – програміст взаємодіє з уніфікованими програмними інтерфейсами, а не з низькорівневими регістрами апаратури. Це не лише стандартизує код, полегшуючи його перенесення між різними платформами, а й дозволяє зосередитися на бізнес-логіці пристрою, оскільки більшість стандартних завдань, як-от керування драйверами, вже вирішені розробниками ядра системи. Крім того, наявність ОС гарантує вищий рівень надійності та безпеки, забезпечуючи багатозадачність, де збір даних, обробка сигналів та мережевий обмін відбуваються без взаємних конфліктів.

Операційні системи прийнято розділяти на два ключові типи: системи загального призначення (GPOS) та системи реального часу (RTOS).

Операційні системи загального призначення (зокрема десктопні ОС) спроектовані для максимізації продуктивності та паралельного виконання багатьох

процесів. У таких системах затримка виконання чи реакції на подію в кілька мілісекунд не вважається збоєм.

Натомість **операційні системи реального часу** спеціально розроблені для середовищ, де важлива часова передбачуваність та детермінованість. При застосуванні RTOS кожна дія має чітко визначений часовий інтервал виконання, а система негайно реагує на події завдяки механізму пріоритетів (менш важливі завдання призупиняються, щоб забезпечити виконання пріоритетних процесів). Вибір на користь RTOS є обґрунтованим у проєктах з обмеженими ресурсами, де необхідно поєднувати високу швидкість відгуку з багатопотоковою обробкою даних, забезпечуючи при цьому масштабованість та надійність програмного забезпечення.

Важливо, що при застосуванні ОС змінюється парадигма виконання коду. Традиційний підхід, відомий як «Super Loop», передбачає послідовне виконання команд у безкінечному циклі, де будь-яка затримка в одній частині коду автоматично блокує весь пристрій. Натомість RTOS перетворює програму на сукупність незалежних завдань (Tasks), кожне з яких працює у власному циклі (рис. 8.1). Планувальник RTOS керує цими завданнями, розподіляючи час процесора згідно з їхнім пріоритетом.

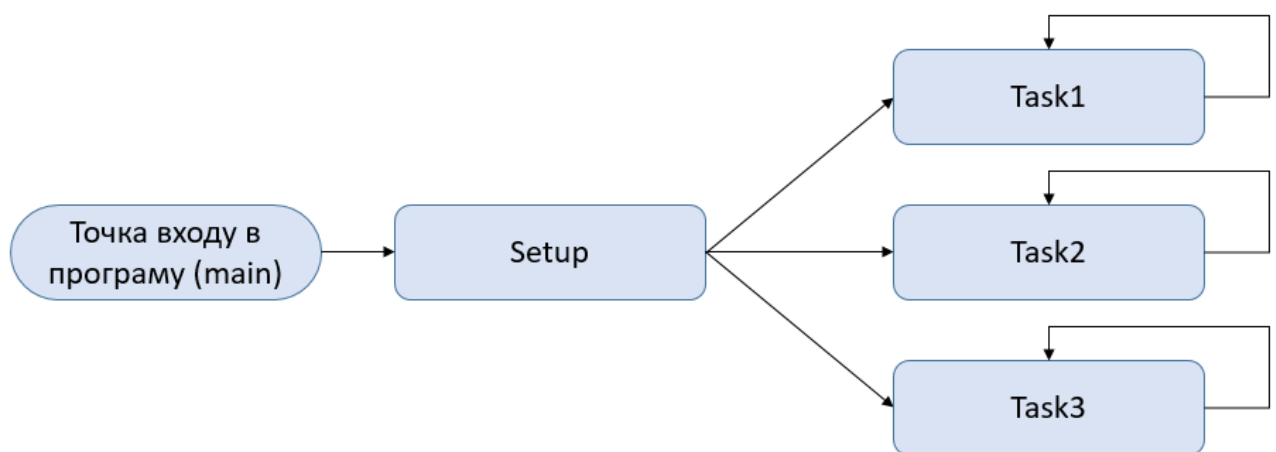


Рис. 8.1. Схема багатозадачного виконання з застосуванням RTOS

У випадку використання багатоядерних мікроконтролерів RTOS відкриває можливості для паралелізму, розподіляючи окремі завдання між фізичними ядрами процесора (рис. 8.2), що значно підвищує продуктивність.

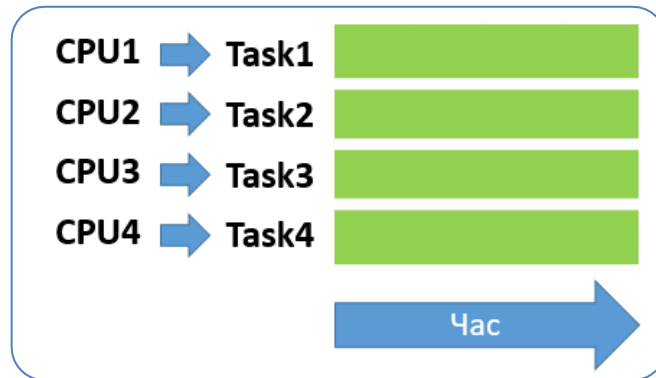


Рис. 8.2. Багатозадачність на чотирьохядерному мікроконтролері

В одноядерних системах процесор фізично виконує лише одну інструкцію в конкретний момент, але завдяки надшвидкому перемиканню контексту між завданнями створюється ілюзія їхнього паралельного виконання (рис. 8.3).

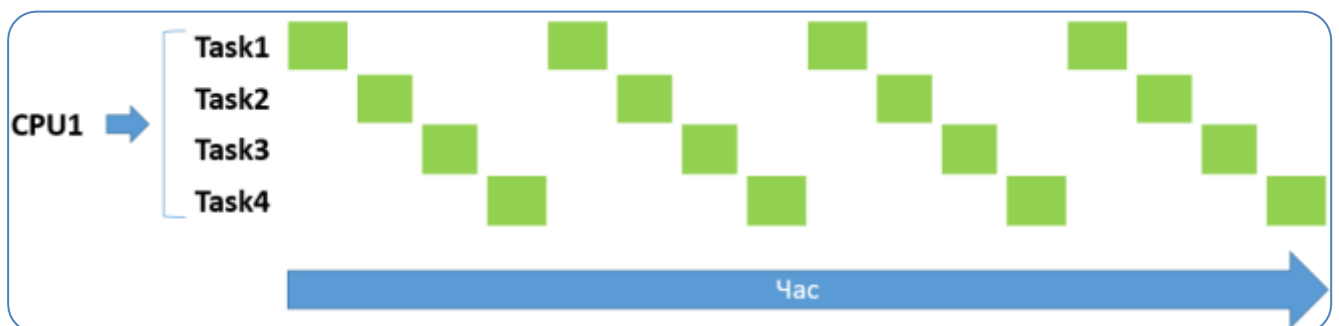


Рис. 8.3. Приклад розподілу процесорного часу (Time-slicing) для моноядерних мікроконтролерів

Інтеграція та налаштування FreeRTOS

Існує велика кількість операційних систем реального часу [15], FreeRTOS є однією з найпопулярніших, завдяки відкритому вихідному коду, мінімалістичному ядру та широкій підтримці різноманітних апаратних архітектур. Завантажити актуальний архів із вихідним кодом системи можна на офіційному сайті: <https://freertos.org/>.

Процес інтеграції RTOS у проєкт зазвичай починається з аналізу завантаженого архіву, який поділено на два ключові каталоги, FreeRTOS та FreeRTOS-Plus. Перша тека містить архітектурно-залежні файли ядра та базові демонстраційні проєкти. Друга тека містить бібліотеки для підтримки мережевих протоколів, файлових систем та засобів безпеки. Для початку роботи необхідно обрати в директорії FreeRTOS/Demo конфігурацію, що відповідає цільовій апаратній платформі та середовищу розробки. Після імпорту файлів до власного проєкту ключовим етапом є редагування параметрів через файл **FreeRTOSConfig.h**, який налаштовує планувальник, методи розподілу пам'яті та стратегії витіснення завдань. Фінальний етап передбачає компіляцію проєкту та верифікацію роботи базової конфігурації на цільовому пристрої.

Важливо зауважити, що у випадку використання мікроконтролерів серії ESP32, описана процедура є зайвою, оскільки виробник інтегрував FreeRTOS безпосередньо у фреймворк розробки (ESP-IDF або Arduino), забезпечивши автоматичне підключення та конфігурацію системи. Таким чином, розробник отримує готову багатозадачну інфраструктуру «з коробки».

Програмний інтерфейс (API) FreeRTOS

Функціональні можливості FreeRTOS структуровані за цільовим призначенням: керування завданнями, синхронізація, організація черг (передача даних), робота з таймерами та управління ресурсами пам'яті. Повна технічна документація доступна на офіційному сайті та в довіднику [16]. Особливості реалізації для платформи ESP32 наведені у документації виробника Espressif [17]. Для виконання завдань цієї роботи пріоритетним є опанування функцій створення та управління завданнями.

xTaskCreate() – створює нову задачу, виділяючи для неї окремий стек пам'яті та встановлюючи пріоритет. Ключовими аргументами є вказівник на функцію задачі (**pvTaskCode**), розмір стека (**usStackDepth**) та рівень пріоритету (**uxPriority**). Функція повертає дескриптор, який у подальшому використовується для керування життєвим циклом задачі. Детальний опис параметрів наведено в довіднику [16].

xTaskCreatePinnedToCore() – розширений аналог **xTaskCreate()** для ESP32, що дозволяє закріпити виконання конкретної задачі за певним ядром процесора.

Приклад створення задачі для ESP32:

```
xTaskCreatePinnedToCore(
    SensorRead,           // Функція задачі
    "SensorRead",         // Ідентифікатор (для дебагінгу)
    10000,                // Розмір стека
    NULL,                 // Параметри, що передаються
    1,                    // Пріоритет
    &SensorReadHandle,    // Дескриптор
    0                     // Номер ядра (0 або 1)
);
```

Для управління станом задач використовуються функції **vTaskSuspend()** (призупинення) та **vTaskResume()** (відновлення) [16]. Вони дозволяють динамічно змінювати активність процесів, базуючись на дескрипторі задачі.

На відміну від стандартної функції **delay()**, яка блокує роботу всього мікроконтролера, функція **vTaskDelay()** переводить поточну задачу у стан очікування на задану кількість тактів, дозволяючи планувальнику RTOS перемикнути на виконання інших, більш пріоритетних завдань. Для коректного переведення часу з мілісекунд у системні такти рекомендується використовувати **pdMS_TO_TICKS(ms)** або ділення на **portTICK_PERIOD_MS**:

```
vTaskDelay(pdMS_TO_TICKS(1000)); // Затримка на 1000 мс
```

В FreeRTOS будь-яка задача реалізується як звичайна функція. Однак, щоб FreeRTOS могла успішно керувати цією функцією як окремою «задачею», вона повинна відповідати трьом вимогам:

- 1) Функція повинна мати тип повернення **void** і один параметр – вказівник типу **void ***, який дозволяє передавати в задачу будь-які дані (конфігураційні структури, посилання на об'єкти тощо) при її створенні.

Навіть якщо задача не потребує вхідних даних, цей параметр є обов'язковим.

- 2) Тіло функції обов'язково повинно бути загорнуте у нескінченний цикл **for(;;)** або **while(1)**, оскільки планувальник ОС очікує, що завдання постійно виконує корисну роботу або перебуває в стані блокування.
- 3) Функція ніколи не повинна повертати значення (інструкція **return** заборонена). Якщо виконання функції завершиться, ядро операційної системи втратить контроль над потоком і це призведе до критичної помилки.

Нижче наведено приклад правильного оформлення функції задачі:

```
// Функція задачі: void, назва, вказівник на параметри
void SensorRead(void *pvParameters)
{
    // Отримання та приведення типу параметрів (якщо вони передані)
    // int *sensorPin = (int *)pvParameters;

    for(;;) // Нескінченний цикл – серце задачі
    {
        // 1. Виконання корисної логіки
        int sensorValue = analogRead(A0);

        // 2. Взаємодія з іншими задачами або виведення результатів
        Serial.println(sensorValue);

        // 3. Блокування задачі (vTaskDelay) – обов'язковий крок,
        // що дозволяє планувальнику перемикатися на інші задачі
        vTaskDelay(pdMS_TO_TICKS(100));
    }

    // Код після циклу ніколи не буде виконаний
}
```

Важливо! Якщо написати цикл без затримок, задача «забере» весь час процесора, повністю заблокувавши виконання інших задач, що суперечить принципам багатозадачності.

Примітка. При адаптації фреймворку Arduino для ESP32 розробники прийняли архітектурне рішення – обгорнути звичний для Arduino цикл **loop()** у FreeRTOS-задачу. Тому підпрограма **loop()** – це частина окремої задачі FreeRTOS, що функціонує під управлінням планувальника на рівні з іншими задачами. Саме ця задача виконує **setup()** один раз, а потім заходить у вічний цикл, де викликає код розміщений в **loop()**. Для запобігання конфліктам, оптимальним підходом є перенесення прикладної логіки у власні завдання з чітко визначеними пріоритетами, залишаючи **loop()** порожнім або мінімально завантаженим.

Акустичні випромінювачі та генерація звуку

П'єзоелектричний випромінювач (п'єзодинамік) – це електроакустичний пристрій, робота якого базується на зворотному п'єзоелектричному ефекті. Під дією прикладеної змінної напруги п'єзоелемент зазнає механічних деформацій (вібрацій), що змушує навколишнє повітря коливатися, утворюючи звукові хвилі. Завдяки простоті конструкції та низькому енергоспоживанню, ці елементи є стандартним рішенням для реалізації звукової індикації у вбудованих системах.

Для програмного керування такими випромінювачами можуть використовуватись наступні функції:

- **tone(вивід, частота)** – ініціює генерацію сигналу з вказаною частотою (у Герцах).
- **tone(вивід, частота, тривалість)** – генерує сигнал протягом заданого інтервалу (у мілісекундах), після чого автоматично зупиняє його.
- **noTone(вивід)** – примусово вимикає генерацію звуку.

Функція **tone()** працює на рівні апаратних таймерів, тому вона не зупиняє виконання програми (не блокує процесор). Однак, використання стандартної функції **delay()** разом із **tone()** – це помилка в контексті багатозадачності, тому при генерації мелодій слід обов’язково використовувати **vTaskDelay()**, що дозволить планувальнику перемикатися на інші задачі в паузах між нотами.

```
void TaskPlaySound(void *pvParameters)
{
    pinMode(PIEZO_PIN, OUTPUT);
    for(;;)
    {
        // Початок відтворення сигналу 1 кГц
        tone(PIEZO_PIN, 1000);

        // Використовуємо vTaskDelay, щоб не блокувати інші задачі RTOS
        vTaskDelay(pdMS_TO_TICKS(500));
        // Зупинка генерації
        noTone(PIEZO_PIN);

        // Пауза перед наступним сигналом
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}
```

Завдання

1. Створити основну задачу «Музичний плеєр», яка становить логічне ядро системи. Вона має керувати процесом відтворення мелодій.
2. За допомогою окремої задачі реалізувати обробку натискань кнопок. Кнопки «Rewind» та «Forward» мають перемикати поточну мелодію, а кнопка «Pause» – призупиняти відтворення (з можливістю подальшого відновлення за допомогою «Play»).

3. Реалізувати задачу для оновлення дисплея, яка виводить текстову назву мелодії, що відтворюється в даний момент.
4. Створити задачу для обробки сигналів джойстика, яка забезпечує переміщення графічного символу «X» по екрану у всіх напрямках (одночасно з відтворенням музики та виведенням тексту).

Порядок виконання роботи

Крок 1. Створити проєкт для ESP32 у симуляторі Wokwi та зібрати схему згідно з рис. 8.4.

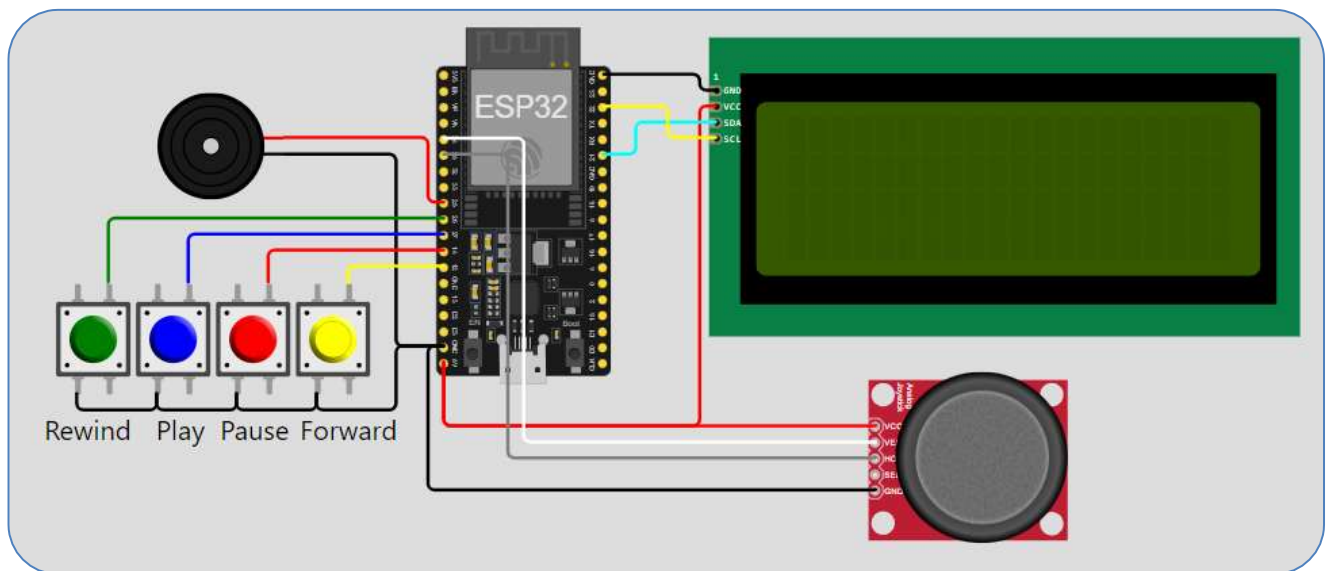


Рис. 8.4. Схема підключення елементів до мікроконтролера ESP32

Крок 2. Додати до проєкту файл наведений приклад коду та встановити необхідні бібліотеки:

```
#include <Arduino.h>
#include <LiquidCrystal_I2C.h>
#include "pitches.h" //Бібліотека AnyRtttl для роботи з нотами

// Адреса I2C дисплею
#define I2C_ADDR 0x27

// Визначення пінів для периферійних пристроїв
#define buzzer 25
```

```

#define rewind 26
#define play 27
#define pause 14
#define forward 12
#define joystick_X 34
#define joystick_Y 35

// Створення об'єкта класу LiquidCrystal_I2C з вказаною адресою та розміром
LiquidCrystal_I2C lcd(I2C_ADDR, 20, 4);

// Ідентифікатор (дескриптор) завдання для відтворення музики
TaskHandle_t MusicPlayerHandle = NULL;

// Прототипи функцій задач та мелодій
void Melody1();
void Melody2();
void Melody3();
void Melody4();
void Joystick(void *pvParameters);
void ButtonCheck(void *pvParameters);
void MusicPlayer(void *pvParameters);

void setup()
{
    Serial.begin(115200);
    Serial.println("Hello, ESP32!");

    // Налаштування кнопок як входів з підтяжкою до VCC
    pinMode(rewind, INPUT_PULLUP);
    pinMode(play, INPUT_PULLUP);
    pinMode(pause, INPUT_PULLUP);
    pinMode(forward, INPUT_PULLUP);

    // Ініціалізація дисплею та виведення тексту

```

```

    lcd.init();
    lcd.backlight();
    lcd.setCursor(0, 0);
    lcd.print("0ooo00oo000000oooo0oo00oooo00ooo00oo000000oooo0oo00oooo00ooo00
oo000000oooo0oo00oooo0");

    // Створення задач (розподіляються між двома ядрами ESP32)
    // MusicPlayer працює на ядрі 0
    xTaskCreatePinnedToCore(MusicPlayer, "MusicPlayer", 10000, NULL, 1,
&MusicPlayerHandle, 0);
    // Призупинення виконання таска MusicPlayer
    vTaskSuspend(MusicPlayerHandle);

    // Обробка натискання кнопок та джойстика відбуваються на ядрі 1
    xTaskCreatePinnedToCore(ButtonCheck, "ButtonCheck", 5000, NULL, 1, NULL, 1);
    xTaskCreatePinnedToCore(Joystick, "Joystick", 5000, NULL, 1, NULL, 1);
}

// Функція завдання відтворення мелодій
void MusicPlayer(void *pvParameters)
{
    while (true)
    {
        Melody1();
        Melody2();
        Melody3();
        Melody4();
    }
}

// Функція для обробки натискання кнопки "Play"
void ButtonCheck(void *pvParameters)
{
    // Нескінченний цикл

```

```

while (true)
{
    if (!digitalRead(play))          // Опитування стану кнопки
    {
        vTaskResume(MusicPlayerHandle); // Поновлення виконання задачі
        Serial.println("Play");

        vTaskDelay(pdMS_TO_TICKS(300)); // Уникнення вібрації контактів
    }

    vTaskDelay(pdMS_TO_TICKS(10));    // Затримка для перемикання контексту
}

// Функція управління курсором через джойстик
void Joystick(void *pvParameters)
{
    // Початкові координати курсора
    byte posX = 10, posY = 2;
    byte lastPosX = posX;
    int valueX = 0;

    // Встановлення курсора на початкову позицію при запуску завдання
    lcd.setCursor(posX, posY);
    lcd.print("X");

    while (true)
    {
        // Зчитування значень з джойстика
        valueX = analogRead(joystick_X);

        // Логіка навігації курсором
        if (valueX < 100 && posX > 0) { posX--; }
        if (valueX > 4000 && posX < 19) { posX++; }
    }
}

```

```

// Оновлення курсора лише при зміні його позиції
if (posX != lastPosX)
{
    lcd.setCursor(lastPosX, posY);
    lcd.print(" "); // Очищення попередньої позиції курсора
    lcd.setCursor(posX, posY);
    lcd.print("X"); // Виведення курсора в новій позиції
    lastPosX = posX;
}

vTaskDelay(pdMS_TO_TICKS(100)); // Періодичність опитування джойстика
}
}

// Основний цикл порожній, всі функції виконуються в окремих завданнях
void loop() {}

void Melody1()
{
    // Створити/імпортувати послідовність нот для першої композиції
    // Використовувати функції tone() для відтворення окремих нот
    // Приклад: tone(buzzer, NOTE_C4, 500);
    // Додати затримку між нотами, щоб мелодія була ритмічною
    // Приклад: vTaskDelay(pdMS_TO_TICKS(500));
}

void Melody2()
{
    // Створити/імпортувати послідовність нот для другої композиції
}

void Melody3()
{

```

```

    // Створити/імпортувати послідовність нот для третьої композиції
}

void Melody4()
{
    // Створити/імпортувати послідовність нот для четвертої композиції
}

```

Логіка роботи програми базується на принципах багатозадачності операційної системи реального часу. Замість виконання всього коду послідовно в одному циклі **loop()**, система розділена на незалежні задачі. На початку коду відбувається підготовка системи до роботи: встановлюються режими роботи пінів (вхід/вихід) та ініціалізується дисплей; реєструються три незалежні задачі (**MusicPlayer**, **ButtonCheck**, **Joystick**). Використання функції **xTaskCreatePinnedToCore** дозволяє розподілити навантаження. Задачі інтерфейсу (кнопки та джойстик) працюють на одному ядрі, а процес відтворення музики – на іншому, що мінімізує вплив обробки графіки на стабільність звукового сигналу. Задача плеєра одразу призупиняється (**vTaskSuspend**). Тобто, відтворення не почнеться автоматично, доки не надійде сигнал від натискання кнопки «Play».

Функція відтворення музики (**MusicPlayer**) – це «виконавчий механізм» плеєра. Вона викликає функції відтворення мелодій (**Melody1-Melody4**) по черзі, в нескінченному циклі **while(true)**. Оскільки задача призупинена при старті, вона не споживає обчислювальні ресурси процесора, доки планувальник не отримає команду **vTaskResume**. Як тільки команда надходить, задача продовжує роботу з того місця, де була зупинена.

Задача обробки кнопок (**ButtonCheck**) – це «обробник подій», який відстежує стан фізичного інтерфейсу. Функція постійно опитує стан пінів, до яких підключені кнопки. При виявленні падіння напруги на піні (натискання кнопки **Play**), виконується команда **vTaskResume(MusicPlayerHandle)**, що переводить задачу плеєра зі стану «призупинено» у стан «виконується». Використання затримки

vTaskDelay дозволяє уникнути помилкових спрацьовувань через вібрацію контактів кнопки.

Задача керування джойстиком (**Joystick**) – це «інтерактивний інтерфейс», що зчитує аналогові значення напруги, перетворює їх на координати і оновлює позицію символу «X» на екрані. Логіка програми передбачає оновлення дисплея лише тоді, коли координати змінилися, що запобігає мерехтінню дисплея та зайвому навантаженню на шину I2C. Зверніть увагу, що в даному прикладі задача **Joystick** здійснює прямий запис на дисплей. У складніших системах, де кілька завдань мають доступ до одного ресурсу (дисплея, шини I2C чи UART), виникає ризик конфліктів доступу (race condition), що призводить до відображення некоректних даних. Для вирішення цієї проблеми в наступній роботі будуть вивчатися механізми синхронізації (м'ютекси/семафори).

Крок 3. Використовуючи функцію **tone()**, реалізувати логіку відтворення різних звукових сигналів у підпрограмах **Melody1**, **Melody2**, **Melody3** та **Melody4**.

Крок 4. Доповнити функцію **ButtonCheck** кодом опитування решти кнопок (rewind, pause, forward) або створити окремі функції для цього. Забезпечити коректну обробку станів, щоб уникнути конфліктів між командами.

Крок 5. До функції **MusicPlayer** додати логіку виведення інформації про поточну мелодію на LCD-дисплей. Для цього кожній мелодії присвоїти окрему назву та забезпечити виклик методу **lcd.print()** у момент зміни мелодії, попередньо очистивши відповідний рядок на екрані.

Крок 6. Доповнити код у задачі **Joystick**, для реалізації переміщення символу «X» у всіх напрямках. Рішення про напрямок переміщення необхідно приймати на основі значень зчитаних з пінів **joystick_X** та **joystick_Y**.

Порядок подання звіту

1. Виконайте вхід до курсу на платформі дистанційного навчання «Сікорський».
2. Перейдіть до розділу, що відповідає поточній роботі та відкрийте форму для подання результатів.
3. У полі для текстової відповіді додайте гіперпосилання на збережений проєкт у середовищі Wokwi.

Важливо! Перед відправленням переконайтеся у працездатності проєкту запустивши симуляцію!

4. Збережіть відповідь для внесення проєкту до бази перевірки викладачем.
-

Контрольні запитання

1. Що таке операційні системи та які вони бувають?
2. Чим відрізняються RTOS від GPOS?
3. Як реалізується багатозадачність на багато- та одноядерному процесорі?
4. Які переваги дає розділення програми на окремі задачі порівняно з написанням усього коду в одному циклі?
5. Як створити окрему задачу з використанням FreeRTOS?
6. Для чого використовується функція `xTaskCreatePinnedToCore` і що означає параметр ядра? Чому важливо розподіляти задачі на різні ядра?
7. Що обов'язково повинна містити кожна задача?
8. У чому полягає принципова відмінність між стандартною затримкою `delay()` та `vTaskDelay()`?
9. Що відбудеться, якщо в одній із задач забути викликати `vTaskDelay()`?
10. Чому у проєктах з використанням FreeRTOS функція `loop()` зазвичай залишається порожньою?
11. Як реалізується програмне керування життєвим циклом задачі (зупинка та поновлення)?

ПРАКТИЧНА РОБОТА № 9

СИНХРОНІЗАЦІЯ ДОСТУПУ ДО СПІЛЬНИХ РЕСУРСІВ В RTOS

Мета: Навчитися реалізовувати безпечну взаємодію між паралельними процесами та впроваджувати механізми блокування для захисту спільних ресурсів.

План проведення практичного заняття

- Опрацювання теоретичних матеріалів.
- Покрокове виконання завдань.
- Подання результатів (звіту) на перевірку.

Теоретичні відомості

Механізми синхронізації у RTOS застосовуються для узгодження роботи задач та забезпечення цілісності даних при спільному використанні ресурсів. Без належної синхронізації виникають конфлікти доступу та втрата даних, що призводить до нестабільної роботи системи. У FreeRTOS до їх складу входять черги та семафори (м'ютекси, бінарні, з лічильником та рекурсивні [16]).

Передача даних між задачами

Черга (Queue) – це структура даних призначена для безпечного асинхронного обміну даними між задачами, що працює за принципом FIFO (First In, First Out), де кожен новий елемент додається в кінець, а вилучається з початку (рис. 9.1).

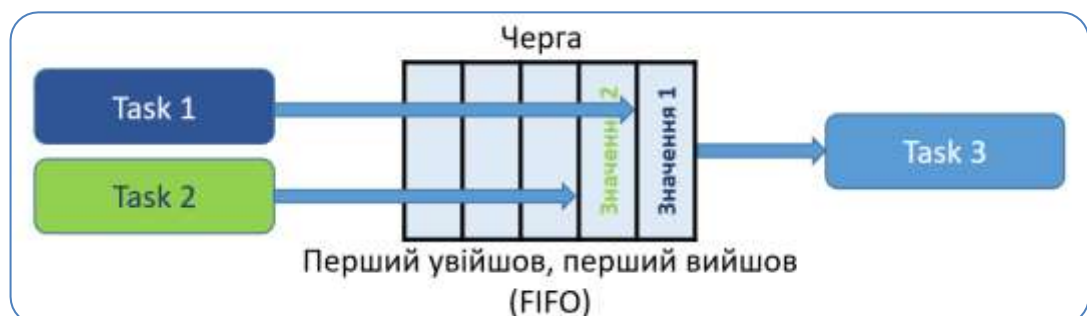


Рис. 9.1. Схема обміну даними між задачами через чергу

Використання черг замість глобальних змінних значно підвищує безпеку системи. Глобальні змінні створюють ризик спотворення інформації при одночасному записі з кількох задач, а також призводять до втрати даних, якщо швидкість генерації інформації перевищує швидкість її обробки.

Створення черги виконується функцією `xQueueCreate(length, itemSize)` [16], яка виділяє необхідну пам'ять та повертає дескриптор `QueueHandle_t`. Функція приймає два аргументи: загальну кількість елементів (довжину черги) та розмір кожного елемента в байтах. Якщо не вдається виділити потрібний об'єм пам'яті, функція повертає значення `NULL`. Приклад створення черги для п'яти цілих чисел виглядає наступним чином:

```
QueueHandle_t myQueue = xQueueCreate(5, sizeof(int));
```

Відправка даних до черги реалізується функцією `xQueueSend` [16]. Вона потребує дескриптора черги, вказівника на дані та часу очікування. Встановлення часу очікування за допомогою константи `portMAX_DELAY` вказує системі на необхідність очікувати необмежений час, доки в черзі не з'явиться вільне місце. Якщо черга заповнена, функція повертає `errQUEUE_FULL`, а при успішному записі – `pdTRUE`. Приклад відправки цілого числа в чергу `myQueue`:

```
int someValue = 88;

xQueueSend(myQueue, &someValue, portMAX_DELAY);
```

Для отримання даних з черги застосовується `xQueueReceive` [16]. Вона копіює дані у вказану змінну, після чого видаляє їх з черги. При успішному отриманні даних з черги, функція повертає `pdTRUE`, в іншому випадку – `pdFALSE`. Якщо черга порожня, аргумент `portMAX_DELAY` змушує чекати на появу нових даних. Приклад:

```
int someValue2;

xQueueReceive(myQueue, &someValue2, portMAX_DELAY);
```

Після завершення роботи з чергою її можливо видалити функцією `vQueueDelete` [16], щоб звільнити оперативну пам'ять. Для цього слід передати в функцію дескриптор відповідної черги.

Керування доступом до спільних ресурсів

Семафори є основними механізмами синхронізації у FreeRTOS, що дозволяють координувати виконання задач та керувати доступом до спільних ресурсів. Виділяють чотири ключові типи цих об'єктів: м'ютекс, семафор з лічильником, бінарний семафор та рекурсивний семафор. Вибір конкретного типу залежить від того, чи потрібно забезпечити ексклюзивний доступ, сигналізувати про подію або керувати груповим доступом.

М'ютекс (від англ. **mutual exclusion**) використовується для забезпечення ексклюзивного доступу до ресурсу, запобігаючи одночасному виконанню критичних ділянок коду різними задачами. На практиці м'ютекс застосовується як «ключ» до ресурсу, який необхідно захопити перед початком використання і обов'язково повернути після завершення. Задача, яка успішно захоплює м'ютекс, стає його єдиним власником і лише вона має повноваження його повернути. Якщо інша задача намагається отримати доступ до цього ж м'ютексу, вона переводиться у стан очікування.

Для використання цього механізму спочатку необхідно створити м'ютекс за допомогою функції `xSemaphoreCreateMutex`. Захоплення м'ютексу виконується перед виконанням критичної ділянки коду (напр. доступ до змінної, відправка даних в мережу, оновлення дисплея тощо) функцією `xSemaphoreTake(handle, ticksToWait)`, де `ticksToWait` вказує час очікування, якщо м'ютекс зайнятий іншою задачею. Після виконання критичної ділянки коду в задачі викликається `xSemaphoreGive(handle)`, повертаючи м'ютекс у вільний стан. Важливою перевагою м'ютексів є підтримка механізму успадкування пріоритетів. Якщо високопріоритетна задача очікує на м'ютекс, який утримує задача з низьким пріоритетом, система тимчасово підвищує пріоритет останньої, щоб вона швидше вивільнила м'ютекс.

Нижче наведено приклад застосування м'ютекса для захисту виведення даних у послідовний порт:

```
#include <Arduino.h>

SemaphoreHandle_t serialMutex = xSemaphoreCreateMutex(); // Створення м'ютекса

// Функція задачі, яка приймає ім'я задачі як параметр
void taskPrinter(void *pvParameters)
{
    char* taskName = (char*)pvParameters;

    while (1)
    {
        // Захоплення м'ютексу або очікування якщо він зайнятий
        if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE)
        {
            // --- ПОЧАТОК КРИТИЧНОЇ СЕКЦІЇ ---
            // Завдяки м'ютексу інша задача не зможе виконувати цю секцію паралельно
            Serial.print("Повідомлення від: ");
            char* str = taskName;
            while (*str != '\0')
            {
                Serial.print(*str);
                delay(1); // Штучна пауза для розтягнення часу виконання
                str++;
            }
            Serial.println();
            // --- КІНЕЦЬ КРИТИЧНОЇ СЕКЦІЇ ---

            // Звільнення м'ютексу
            xSemaphoreGive(serialMutex);
        }
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}
```

```

    }
}

void setup()
{
    Serial.begin(115200);
    // Створення двох задач, що працюють з одним ресурсом
    xTaskCreate(taskPrinter, "Task A", 2048, (void*)"Task A", 1, NULL);
    xTaskCreate(taskPrinter, "Task B", 2048, (void*)"Task B", 1, NULL);
}

void loop() { }

```

Планувальник RTOS може перервати виконання задачі посеред операції виведення даних. Якщо інша задача розпочне виведення у той самий момент, дані змішаються, що призведе до спотворення виводу («каші» в серійному моніторі). В прикладі вище, процес виведення символів спеціально сповільнено, щоб планувальник гарантовано встиг переключити задачі посередині виведення.

Якщо повідомлення коротке, воно потрапляє в апаратний буфер UART майже миттєво. Весь процес запису може завершитися швидше, ніж планувальник встигне переключити контекст на іншу задачу. Ситуація, коли програма «працює справно і без м'ютекса», є оманливою та не означає, що код безпечний. Поведінка такої системи є недетермінованою та залежить від того, наскільки швидко працює процесор, як часто перемикаються задачі та наскільки довгі рядки слід виводити.

Завдання

1. Створіть три задачі: **taskDataGenerator1**, **taskDataGenerator2** та **taskDataMonitor**. Перші дві – повинні генерувати випадкові числа (0–99) та кожні 500 мс відправляти їх у чергу розміром 10 елементів. Друга задача має постійно очікувати дані з цієї черги за допомогою **xQueueReceive** і виводити їх у серійний монітор.

2. Створіть чотири ідентичні задачі **taskRowAnimator**, кожна з яких керує анімацією символу «-» в окремому рядку LCD-дисплея (рядки 0, 1, 2, 3). При створенні задачі через параметр **pvParameters** передавайте індекс рядка та номер аналогового піна для потенціометра. Задача повинна зчитувати значення потенціометра, перетворювати його на затримку і рухати символ по своєму рядку (позиції 0–19). Дійшовши до кінця, символ має «зникати» та починати рух спочатку.
3. Щоб уникнути графічних артефактів (накладання тексту, мерехтіння) через конкурентний доступ чотирьох задач до одного дисплея реалізуйте захоплення м'ютекса (**xSemaphoreTake**) перед викликом методів **lcd.setCursor()** та **lcd.print()** та його вивільнення (**xSemaphoreGive**) одразу після завершення операції.
4. Використовуючи функцію **uxTaskGetStackHighWaterMark**, дослідіть використання стека для кожної задачі. Підберіть мінімальний розмір стека при створенні задачі (**xTaskCreate**), при якому система працює стабільно.

Додаткове завдання. Створіть лічильний семафор із початковим значенням «3». Змініть логіку задач **taskRowAnimator** так, щоб перед початком циклу руху символу по рядку, задача мала отримати семафор (**xSemaphoreTake**). Якщо доступних семафорів немає, задача має блокуватися. Після того, як символ досягає правої межі дисплея (позиція 19), задача повинна звільнити семафор (**xSemaphoreGive**), дозволяючи іншій задачі розпочати роботу.

Важливо! Бали за додаткові завдання нараховуються за умови вчасного подання звіту.

Порядок виконання роботи

Крок 1. Створити новий проєкт, додати бібліотеку **LiquidCrystal_I2C** та зібрати електричну схему згідно з рисунком 9.2.

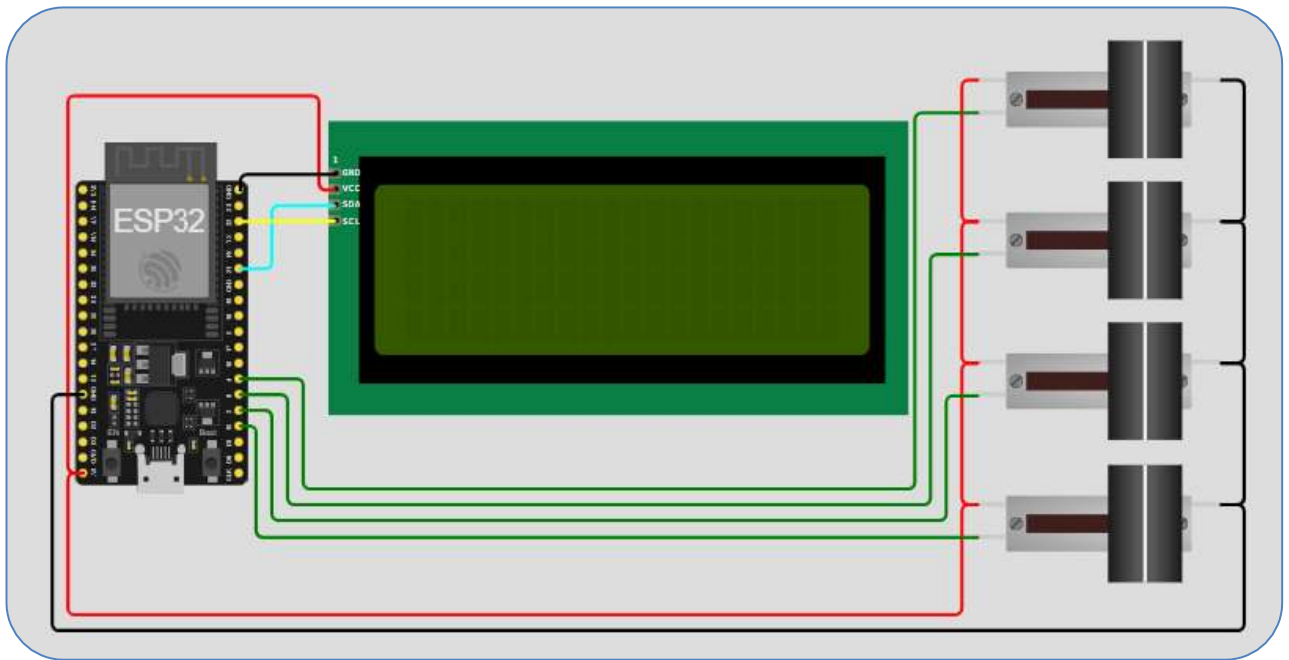


Рис. 9.2. Схема

Крок 2. Імпортувати наступний приклад коду, проаналізувати та перевірити його працездатність:

```
#include <Arduino.h>
#include <LiquidCrystal_I2C.h>

LiquidCrystal_I2C lcd(0x27, 20, 4);          // Ініціалізація дисплея

// Задача анімації: керує рухом символу в рядку
void taskRowAnimator(void *pvParameters)
{
    int row = (int)pvParameters; // Отримуємо номер рядка через параметр
    int col = 0;
    int oldPos = 0;
    int pin;

    // Прив'язка GPIO до рядка (приклад)
    switch(row)
    {
        case 0: pin = 4; break;
```

```

        case 1: pin = 0; break;
    }
    while (true)
    {
        lcd.setCursor(col, row);
        lcd.print("-");
        lcd.setCursor(oldPos, row);
        lcd.print(" ");

        oldPos = col;
        col = (col + 1) % 20; // Циклічний рух по рядку

        vTaskDelay(pdMS_TO_TICKS((analogRead(pin)/10)));
    }
}

// Генерація даних
void taskDataGenerator1(void *pvParameters)
{
    while(true)
    {
        int data = random(0, 99);

        // TODO: додати відправку data в чергу

        vTaskDelay(pdMS_TO_TICKS(10));
    }
}

// Моніторинг даних
void taskDataMonitor(void *pvParameters)
{
    while(true)
    {

```

```

    int receivedData;

    //TODO: додати отримання даних з черги

    Serial.print("Отримано дані: ");
    Serial.println(receivedData);
    vTaskDelay(pdMS_TO_TICKS(500));
}
}

void setup()
{
    Serial.begin(115200);
    lcd.init();
    lcd.backlight();

    // Створення задач
    xTaskCreate(taskRowAnimator, "Row0", 2048, (void*)0, 1, NULL);
    xTaskCreate(taskDataGenerator1, "Generator1", 2048, NULL, 1, NULL);
    xTaskCreate(taskDataMonitor, "Monitor", 2048, NULL, 1, NULL);
}

void loop()
{
}

```

Крок 3. Реалізувати передачу даних між **taskDataGenerator1**, **taskDataGenerator2** та **taskDataMonitor**. Для цього створити чергу (за допомогою **xQueueCreate**), налаштувати у **taskDataGenerator1**, **taskDataGenerator2** відправку даних в чергу (функцією **xQueueSend**), а в **taskDataMonitor** забезпечити їх отримання (функцією **xQueueReceive**) та вивід.

Крок 4. Створити чотири паралельні задачі, кожна з яких керує відображенням символу «-» у відповідному рядку дисплея. Швидкість руху кожного символу має

залежати від значень, що зчитуються з відповідних аналогових входів (потенціометрів). Запустити програму та змінюючи положення потенціометрів, спостерігати за роботою дисплея. Зафіксувати виникнення графічних артефактів, які виникають внаслідок конкурентного доступу кількох задач до дисплею.

Крок 5. Усунути виявлені артефакти шляхом впровадження механізму синхронізації. Створити м'ютекс за допомогою `xSemaphoreCreateMutex`. У кожній задачі, що працює з дисплеєм, додати виклик `xSemaphoreTake` перед критичною секцією (оновленням екрана) та `xSemaphoreGive` одразу після її завершення, щоб забезпечити ексклюзивний доступ до спільного ресурсу.

Крок 6. Визначити та встановити мінімально необхідний розмір стеку для кожної задачі. Перевірити коректність виконання програми після оптимізації.

Порядок подання звіту

1. Виконайте вхід до курсу на платформі дистанційного навчання «Сікорський».
2. Перейдіть до розділу, що відповідає поточній роботі та відкрийте форму для подання результатів.
3. У полі для текстової відповіді додайте гіперпосилання на збережений проєкт у середовищі Wokwi.

Важливо! Перед відправленням переконайтеся у працездатності проєкту запусивши симуляцію!

4. Збережіть відповідь для внесення проєкту до бази перевірки викладачем.

Контрольні запитання

1. Що таке «спільний ресурс» у багатозадачній системі?
2. Що таке синхронізація та які механізми існують в FreeRTOS?

3. До яких негативних наслідків призводить доступ до даних без використання механізмів синхронізації?
4. Яке призначення черг у FreeRTOS?
5. Що означає термін «м'ютекс»?
6. Яка задача має право «звільнити» м'ютекс?
7. Що відбувається із задачею, яка намагається викликати функцію **xSemaphoreTake** для вже зайнятого м'ютекса?
8. Як перевірити в коді, чи вдалося задачі захопити м'ютекс після виклику **xSemaphoreTake**?

ПРАКТИЧНА РОБОТА № 10

ОПТИМІЗАЦІЯ ПРОГРАМИ ЗА ДОПОМОГОЮ ПРЯМОГО ЗВЕРНЕННЯ ДО РЕГІСТРІВ МІКРОКОНТРОЛЕРА

Мета: Опанувати принцип прямого керування апаратними портами (GPIO) мікроконтролера через його регістри для досягнення максимальної швидкості та ефективності виконання коду.

План проведення практичного заняття

- Опрацювання теоретичних матеріалів.
- Покрокове виконання завдань.
- Подання результатів (звіту) на перевірку.

Теоретичні відомості

Порти вводу/виводу ATmega328P

Взаємодія мікроконтролера ATmega328P із периферійними пристроями реалізується через порти вводу/виводу (рис. 10.1). Ці порти функціонують як керовані набори бітів, стан кожного з яких відповідає рівню сигналу на фізичному виводі (піні) мікроконтролера. Кожен біт може бути програмно налаштований для передачі або прийому даних.

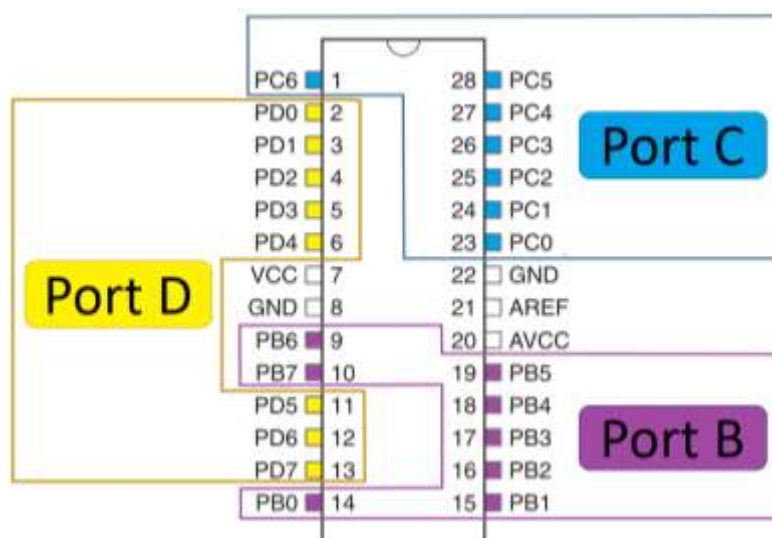


Рис. 10.1. Розташування портів та виводів ATmega328P (DIP-28)

Архітектура контролера передбачає наявність трьох портів: PORTB, PORTC та PORTD. Ідентифікація конкретного виводу базується на його позначенні, де літера вказує на приналежність до порту, а цифра – на порядковий номер біта (наприклад, PD1 означає перший біт порту D, PC4 – четвертий біт порту C).

При розробці програмного забезпечення для плат Arduino Uno необхідно враховувати різницю між фізичною нумерацією виводів самого мікроконтролера та маркуванням роз'ємів на платі розробника. Порт PORTD охоплює вісім ліній (PD0-PD7), які відповідають цифровим виводам 0-7 плати. Порт PORTB також містить вісім ліній – виводи PB0-PB5 з'єднані з цифровими пінами 8-13, тоді як PB6 та PB7 використовуються для підключення зовнішнього кварцового резонатора. Порт PORTC включає сім ліній (PC0-PC6), які на платі відповідають аналоговим пінам A0-A5 та лінії скидання (RESET).

Регістри для програмного керування портами

Для керування портами мікроконтролера ATmega328P використовуються три типи спеціалізованих 8-бітних регістрів для кожного порту (x – B, C, D):

- **DDRx** (Data Direction Register) – визначає напрямок передачі даних. Якщо біт встановлено в 1, відповідний вивід працює на вихід (Output), якщо 0 – на вхід (Input).
- **PORTx** – використовується для встановлення логічного стану виводу. Якщо вивід налаштований як вихід, 1 формує рівень HIGH, а 0 – LOW. Якщо вивід налаштований як вхід, запис 1 активує внутрішній підтягувальний резистор (Pull-up), а 0 – відключає його.
- **PINx** – дозволяє зчитувати поточний логічний рівень на виводах порту в режимі реального часу, незалежно від напрямку передачі.

Розглянемо завдання – одночасно засвітити два світлодіоди, підключені до пінів 8 (PB0) та 9 (PB1) Arduino Uno. З використанням прямого звернення до регістрів реалізація такої програми включатиме наступні етапи:

- 1) Для налаштування PB0 та PB1 як виходів, необхідно записати одиниці у відповідні розряди регістра DDRB. Використання префікса 0b дозволяє вказати число у двійковому форматі, де кожен біт відповідає номеру піна (від 7 до 0):

```
DDRB = 0b00000011; // PB1 та PB0 налаштовано як виходи
```

- 2) Після налаштування напрямку слід встановити високий логічний рівень на обраних виводах шляхом запису одиниць у відповідні біти регістра PORTB:

```
PORTB = 0b00000011; // Подача живлення на PB0 та PB1
```

- 3) Регістр PINB дозволяє перевірити стан групи пінів:

```
if (PINB == 0b00000011)
{
    // Дія, якщо стан пінів PB0-PB1 – HIGH, інші – LOW
}
```

Такий спосіб звернення до портів є значно швидшим за стандартні функції **digitalWrite()** чи **digitalRead()**, оскільки він виконується за один такт процесора.

Побітові операції

Побітові операції – це операції, які виконуються безпосередньо над окремими бітами числа. Вони дозволяють змінювати, перевіряти або комбінувати окремі біти без впливу на інші.

Назва «побітові» походить від того, що обчислення відбуваються на рівні бітів – найменших одиниць інформації в комп'ютері. Наприклад, число 13 у двійковому представленні виглядає як 00001101. Побітові операції дозволяють працювати саме з цими нулями та одиницями.

Основні побітові оператори: ~, <<, >>, |, &, ^.

Оператор ~ (NOT) відповідає за інверсію. Він найпростіший, оскільки просто перевертає кожен біт у числі. Нулі стають одиницями, а одиниці – нулями.

Якщо у є число **00000001**, то після операції **~00000001** воно перетвориться на **11111110**. Це часто застосовують для створення «масок» – коли потрібно «заборонити» або скинути лише певні біти, інвертуючи еталонне значення.

Оператори зсуву << та >> (Shift) – це оператори переміщення, які «посувають» біти вліво або вправо. Наприклад, вираз **1 << 3** означає – взяти число 1 (яке в бінарному вигляді виглядає як 00000001) і посунути його на 3 позиції вліво. Результат – 00001000.

Оператор АБО | (OR) – працює за принципом «якщо хоча б десь є одиниця, то в результаті буде одиниця». Він ідеально підходить для того, щоб включити (встановити в 1) конкретний пін, не чіпаючи решту. Якщо до регістра зі станом **00000000** застосувати **| 00000001**, результат буде **00000001**. Старі біти нікуди не діваються, вони просто зберігають свій попередній стан, якщо там вже були одиниці.

Оператор & (AND) – працює суворіше, результат буде одиницею тільки в тому випадку, якщо одиниці є в обох числах одночасно. Наприклад, якщо є регістр **00001010** і необхідно дізнатися, чи встановлений там третій біт, можна виконати наступне: **00001010 & 00001000**. Результат – **00001000**. Оскільки це не нуль, значить біт встановлено.

Оператор Виключне АБО ^ (XOR) – дає одиницю тільки тоді, коли біти в обох числах різні (один – нуль, інший – один). Якщо зробити **регістр ^ 00000001**, то біт у першій позиції переключиться, якщо був 0 – стане 1, якщо був 1 – стане 0. Інші біти залишаться без змін. Ідеально підходить для «блимання» світлодіодами, оскільки не треба знати їх поточний стан, достатньо просто перемикнути на протилежний.

Завдання

1. Відкрити проєкт присвячений блиманню світлодіодами певних кольорів за варіантом (розроблений на першій роботі).

2. Виконати повний рефакторинг коду, замінивши стандартні бібліотечні функції Arduino (такі як `pinMode`, `digitalWrite`, `digitalRead`) на прямі маніпуляції з відповідними регістрами `DDRx`, `PORTx` та `PINx`. Код повинен забезпечувати ідентичну функціональність пристрою, але без використання бібліотечних абстракцій.

Додаткове завдання. Здійсніть портування розробленого алгоритму на іншу платформу, таку як ESP32, STM32 або Raspberry Pi Pico. Опрацюйте технічну документацію (Datasheet) обраного мікроконтролера та реалізуйте аналогічне керування виводами через специфічні для нього регістри.

Важливо! Бали за додаткові завдання нараховуються за умови вчасного подання звіту.

Порядок виконання роботи

Крок 1. Відкрийте програмний проект, розроблений під час першої практичної роботи або зберіть схему відповідно до рисунку 1.7.

Крок 2. Опрацюйте теоретичні відомості та технічну документацію (Datasheet) мікроконтролера ATmega328P, що використовується в платі Arduino Uno. Зосередьте увагу на розділах, присвячених архітектурі портів введення-виведення, щоб ідентифікувати регістри, необхідні для налаштування пінів та подальшого керування їх станом.

Крок 3. Замініть функції фреймворку Arduino (`pinMode`, `digitalWrite` тощо) на операції прямого запису значень у відповідні регістри керування портами.

Крок 4. Реалізуйте механізм часових затримок для візуалізації процесу блимання світлодіодів без залучення функції `delay()`. Для цього створіть власні програмні цикли або процедури затримки, які забезпечать необхідні часові інтервали шляхом виконання порожніх операцій або через лічильники тактів процесора.

Крок 5. Перейдіть до класичної структури програми мови C/C++, замінивши функції `setup()` та `loop()` на точку входу `int main()`.

Крок 6. Проведіть компіляцію, прошивку мікроконтролера та тестування працездатності пристрою.

Порядок подання звіту

1. Виконайте вхід до курсу на платформі дистанційного навчання «Сікорський».
2. Перейдіть до розділу, що відповідає поточній роботі та відкрийте форму для подання результатів.
3. У полі для текстової відповіді додайте гіперпосилання на збережений проєкт у середовищі Wokwi.

Важливо! Перед відправленням переконайтеся у працездатності проєкту запусивши симуляцію!

4. Збережіть відповідь для внесення проєкту до бази перевірки викладачем.
-

Контрольні запитання

1. Що таке регістри мікроконтролера та яку роль вони відіграють у взаємодії з апаратним забезпеченням?
2. Яким чином визначається напрямок передачі даних (вхід чи вихід) для конкретного піна мікроконтролера?
3. Яке призначення регістрів `DDRx`, `PORTx` та `PINx`?
4. Чому пряме звернення до регістрів забезпечує вищу швидкодію порівняно зі стандартними функціями бібліотеки Arduino?
5. Який побітовий оператор оптимально використовувати для встановлення (активації) одиниці в конкретному біті без зміни решти бітів регістра?

6. Поясніть алгоритм виключення (скидання в «0») конкретного біта в регістрі за допомогою операцій I (&) та інверсії (~).
7. Яка роль операторів зсуву (<<, >>) при створенні бітових масок для керування регістрами?
8. У чому полягає перевага використання оператора XOR (^) при реалізації алгоритму блимання світлодіодом?
9. Як реалізувати програмну затримку, не вдаючись до використання стандартної функції `delay()`?
10. Як архітектура мікроконтролера впливає на спосіб керування периферією?
11. В чому недоліки застосування прямого звернення до регістрів в програмі?

ПРАКТИЧНА РОБОТА № 11

ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ОБЧИСЛЕНЬ ТА ЗМЕНШЕННЯ ЧАСУ ВИКОНАННЯ

Мета: Навчитися застосовувати базові прийоми оптимізації програмного коду та експериментально оцінювати їх ефективність.

План проведення практичного заняття

- Опрацювання теоретичних матеріалів.
 - Покрокове виконання завдань.
 - Подання результатів (звіту) на перевірку.
-

Теоретичні відомості

Продуктивність мікроконтролерних систем обмежена тактовою частотою, обсягом пам'яті та швидкістю виконання інструкцій. Тому ефективне програмування полягає не лише у правильності алгоритму, але й у раціональному використанні ресурсів. Оптимізація програмного коду для мікроконтролерів включає застосування декількох прийомів:

- Раціональне використання типів даних та застосування цілочисельної арифметики замість дійсної, коли це можливо (операції з **float** інколи потребують у десятки разів більше часу).
- Усунення зайвих обчислень у циклах. Вирази, що не змінюються в межах циклу, слід обчислювати один раз перед його виконанням.
- Локалізація змінних. Оголошення змінних у найменшій необхідній області видимості – зменшує споживання пам'яті та час доступу.
- Прямі звернення до регістрів та/або застосування асемблерних вставок у критичних ділянках коду, де потрібна максимальна швидкість або прямий контроль над апаратурою.

Ефективне використання типів даних

Мікроконтролери мають обмежені ресурси, тому використання надто «великих» типів для збереження невеликих чисел або невірний вибір формату може призвести до зайвої витрати пам'яті, уповільнення обчислень і навіть переповнення змінних.

Наприклад, на 8-бітних AVR мікроконтролерах тип `int` займає два байти, хоча для чисел від 0 до 255 достатньо одного байта. У такому випадку краще використати `uint8_t`. Якщо потрібно зберегти число від -128 до 127, доречно застосувати `int8_t`:

```
// Зайва витрата пам'яті
int counter = 0; // 2 байти

// Ефективно
uint8_t counter = 0; // 1 байт
```

Ще одна поширена помилка пов'язана з дробовими числами. Наприклад, на AVR тип `float` займає чотири байти, і обчислення з ним відбуваються повільніше, ніж з цілими числами. Часто можна замінити дробові обчислення на цілочисельні, використовуючи відповідний масштаб. Замість того щоб обчислювати напругу як:

```
float voltage = analogRead(A0) * (5.0 / 1024.0);
```

Можна зробити це цілочисельно, перевівши значення у мілівольти:

```
uint16_t raw = analogRead(A0);
uint16_t voltage_mV = raw * 5000 / 1024; // мілівольти
```

Це дозволяє уникнути повільних операцій з плаваючою комою і економить ресурси. Також важливо правильно використовувати логічні змінні. Для прапорців варто застосовувати `bool` замість `int`.

Усунення зайвих обчислень у циклах

Зменшення часу виконання циклів знижує навантаження на процесор, а також дає можливість зекономити енергію у вбудованих системах. Часто у циклах обчислюються одні й ті ж значення або виконуються операції, які можна винести за межі циклу. Кожна непотрібна операція збільшує час виконання програми. У наступному прикладі на кожній ітерації циклу обчислюється константне значення «**3.14159 / 2.0**». Це зайві обчислення, яких можна уникнути, винесши їх поза цикл, так щоб вони обчислювались лише один раз:

Не рекомендовано:	Оптимізований варіант:
<pre>for (int i = 0; i < 1000; i++) { // Множення і ділення // виконуються 1000 разів sum += i * 3.14159 / 2.0; }</pre>	<pre>// Обчислюється один раз float factor = 3.14159 / 2.0; for (int i = 0; i < 1000; i++) { sum += i * factor; }</pre>

Якщо вираз у циклі складається виключно з констант, сучасні компілятори зазвичай виконують оптимізацію автоматично. Проте, якщо у виразі використовуються змінні, значення яких не змінюються всередині циклу це слід робити вручну. Також, якщо у циклі викликаються функції, результат виконання яких залишається незмінним, можна зберігати їх результат один раз до циклу:

Не рекомендовано:	Оптимізований варіант:
<pre>if (analogRead(A0) > 512 && analogRead(A0) < 600) { digitalWrite(LED_BUILTIN, HIGH); }</pre>	<pre>int val = analogRead(A0); if (val > 512 && val < 600) { digitalWrite(LED_BUILTIN, HIGH); }</pre>

Локалізація змінних

Локалізація змінних — це ще один важливий прийом оптимізації, що присвячений використанню змінних у вузькому, необхідному контексті, тобто

обмеженню їх видимості та часу існування, щоб не навантажувати пам'ять і не створювати зайвих конфліктів у програмі.

У мові C/C++ змінні можуть бути глобальними або локальними. Глобальні змінні оголошуються поза функціями і існують протягом усього часу роботи програми. Вони займають пам'ять у статичній області і доступні будь-якій функції в програмі. З першого погляду це зручно, але при цьому зростає ризик помилок. Випадкова зміна значення в одній частині коду може несподівано вплинути на інші частини програми. Локальні змінні оголошуються всередині функцій та існують лише під час виконання цього блоку коду, звільняючи пам'ять після виходу з нього.

Наприклад:

Не рекомендовано:	Оптимізований варіант:
<pre>#include <Arduino.h> // Глобальна змінна float coefficient = 1.5; float calculateValue(float input) { return input * coefficient + 10; } void setup() { Serial.begin(9600); } void loop() { float input = 5.0; float result = calculateValue(input); Serial.println(result); delay(1000); }</pre>	<pre>#include <Arduino.h> float calculateValue(float input) { // локальна змінна, існує лише // під час виконання функції float coefficient = 1.5; return input * coefficient + 10; } void setup() { Serial.begin(9600); } void loop() { float input = 5.0; float result = calculateValue(input); Serial.println(result); delay(1000); }</pre>

У наведеному вище оптимізованому варіанті локальна змінна **coefficient** існує лише під час виконання функції **calculateValue()**. При зміні логіки функції, значення коефіцієнта змінюється тільки локально, без ризику впливу на інші частини програми.

Асемблерні вставки

Асемблерні вставки (inline assembly) – це фрагменти коду на мові асемблера (близька до машинної мови), які безпосередньо вставляються всередину програми на C/C++. Вони дозволяють програмісту вказати компілятору точні інструкції процесора, які потрібно виконати, замість того щоб компілятор сам перекладав C/C++ код у машинні команди.

Хоча компілятор C++ для Arduino (avr-gcc) добре оптимізує код, бувають ситуації, коли асемблерні вставки дають значні переваги:

- **Прямий доступ до апаратного забезпечення.** Асемблер дозволяє безпосередньо маніпулювати регістрами процесора та портами введення-виведення, минаючи рівень абстракції Arduino. Деякі інструкції процесора, наприклад, для керування сном або атомарних операцій, можуть бути недоступні через стандартні функції C++.
- **Точне керування часом.** Асемблерні інструкції виконуються за певну кількість тактових циклів процесора. Це дозволяє створювати надзвичайно точні затримки або послідовності сигналів, що критично для деяких протоколів або керування специфічними пристроями.
- **Підвищення продуктивності.** Для критичних за часом виконання завдань, таких як швидкі математичні обчислення або обробка сигналів, власноруч написаний асемблерний код може бути швидшим, ніж код, згенерований компілятором.

Розробка асемблерних вставок починається з визначення ділянки коду, яку потрібно переписати на мові асемблера, і вибору відповідних інструкцій для конкретної архітектури мікроконтролера, наприклад AVR або ARM. В програму ці

інструкції вставляються за допомогою конструкції `asm` або `__asm__`. При цьому обов'язково потрібно вказати компілятору, які змінні беруть участь у вставці та які регістри можуть змінюватися, щоб уникнути конфліктів. Після вставки доцільно перевірити коректність роботи коду та ефективність, щоб впевнитися, що створене рішення дійсно працює та/або має переваги над рішенням компілятора C/C++.

Синтаксис inline assembly у форматі GCC Extended Asm такий:

```
asm (
    "інструкції ASM з заповнювачами"
    : список_виходів
    : список_входів
    : список_змінених_ресурсів (опціонально)
);
```

Приклад додавання двох чисел за допомогою C++ та асемблерної вставки:

```
#include <Arduino.h>

void setup()
{
    Serial.begin(9600);
    uint8_t a = 5;
    uint8_t b = 10;

    // --- Стандартна операція множення C++ ---
    uint8_t resultCpp = a + b;

    // --- Використання асемблерної вставки ---
    uint8_t resultAsm = 0;
    asm
    (
        "mov r24, %[aval]    \n\t"    // Копіюємо a у r24
        "mov r25, %[bval]    \n\t"    // Копіюємо b у r25
```

```

    "add r24, r25      \n\t"    // Складаємо r24 + r25
    "mov %[res], r24   \n\t"    // Результат у resultAsm
    : [res] "=r" (resultAsm)    // вихідний операнд (resultAsm)
    : [aval] "r" (a), [bval] "r" (b) // вхідні операнди (a, b)
  );

  Serial.print("Результат C++: ");
  Serial.println(resultCpp);
  Serial.print("Результат ASM: ");
  Serial.println(resultAsm);
}

void loop() { }

```

Очікуваний результат:

```

Результат C++: 15
Результат ASM: 15

```

В наведеному вище прикладі:

- **asm** (іноді **__asm__**) – директива, що дозволяє вставити код на мові асемблера безпосередньо всередину функції на C++.
- **mov r24, %[aval]** – інструкція **mov** означає «перемістити значення». Вона копіює значення операнда **[aval]** (змінної **a**) у регістр **r24**. На AVR (Arduino Uno) регістри **r24**, **r25** – це фізичні 8-бітні регістри процесора.
- **mov r25, %[bval]** – копіює **b** у регістр **r25**.
- **add r24, r25** – інструкція додавання. Складає значення регістрів **r24** та **r25** і результат залишає в **r24**.
- **mov %[res], r24** – зберігає результат у змінну C++ **resultAsm**.
- **: [res] "=r" (resultAsm)** – це вихідний операнд (output). **"=r"** означає, що компілятор повинен виділити регістр, який буде містити результат, і

присвоїти його змінній `resultAsm`. `[res]` – це ім'я (мітка), яке використовується у коді замість `%0`.

- `: [aval] "r" (a), [bval] "r" (b)` – це вхідні операнди (input). "r" означає, що компілятор може помістити значення у будь-який регістр. `%[res]`, `%[aval]`, `%[bval]` – це посилання на ці операнди.

Для Arduino Uno (мікроконтролер ATmega328P) всі інструкції асемблера (перелік усіх доступних команд процесору) визначаються в довіднику AVR Instruction Set [18].

До недоліків використання асемблерних вставок можна віднести зниження переносимості коду. Інструкції, які працюють на одному мікроконтролері (наприклад AVR), не запусаються на іншій архітектурі, наприклад ARM чи ESP32. Це означає, що кожен перехід на іншу платформу потребуватиме переписування низькорівневих ділянок програми. Тому код із асемблерними вставками практично завжди «прив'язаний» до конкретного типу процесора.

Ще одна серйозна проблема – ускладнення читання й підтримки коду. Звичайний розробник, який володіє C/C++, може легко зрозуміти логіку програми на рівні функцій, змінних і структур. Але кілька рядків асемблерної вставки одразу перетворюють код у щось непрозоре, де треба знати регістри, формати інструкцій і вплив на прапорці процесора. Через це навіть дрібні зміни в таких ділянках вимагають глибокого аналізу або коментарів. Крім цього, зростає імовірність помилок при неправильному використанні регістрів або пам'яті.

У мові C/C++ компілятор самостійно керує розподілом регістрів і збереженням значень, але він не може «зазирнути» всередину асемблерної вставки, та оптимізувати код, тому часто застосування вставок може зробити програму повільнішою, а не швидшою. Також, стандартні засоби відлагодження (debugger) не завжди можуть відобразити, що відбувається всередині асемблерної вставки, особливо якщо вона модифікує регістри напряду. Це ускладнює аналіз помилок і перевірку правильності виконання програми.

Вимірювання часу виконання обчислень

Універсальним способом вимірювання тривалості обчислень у вбудованих системах є використання лічильника мікросекунд. Функція `micros()` повертає кількість мікросекунд, що минули з моменту запуску програми. Суть методу полягає у фіксації системного часу безпосередньо перед початком досліджуваної ділянки коду та одразу після її завершення. Алгоритм дій:

- 1) Зберегти поточне значення `micros()` у змінну `startTime`.
- 2) Виконати цільовий код (обчислення, сортування, опитування датчика).
- 3) Зберегти поточне значення `micros()` у змінну `endTime`.
- 4) Обчислити різницю: $\Delta T = \text{endTime} - \text{startTime}$.

Приклад коду:

```
unsigned long startTime = micros(); // Фіксуємо старт

// --- Ділянка коду, що вимірюється ---
performComplexCalculations();
// -----

unsigned long endTime = micros(); // Фіксуємо фініш
unsigned long executionTime = endTime - startTime;

Serial.print("Час виконання: ");
Serial.print(executionTime);
Serial.println(" мкс");
```

На відміну від мілісекундних лічильників (`millis()`), `micros()` дозволяє побачити реальну тривалість швидких математичних операцій. Застосування такого методу не потребує глибоких знань архітектури процесора або налаштування складних регістрів таймерів.

Примітка. Слід враховувати, що:

- виклик функції `micros()` займає певний час (кілька тактів процесора).
- якщо під час обчислень виникло апаратне переривання, результат покаже сумарний час (обчислення + час на обробку переривання).
- значення `micros()` не завжди змінюється на одиницю. На багатьох платах крок складає декілька мкс (наприклад, 4 мкс на Arduino Uno з 16 МГц).
- лічильник `micros()` зазвичай базується на 32-бітному числі. Це означає, що приблизно через 70 хвилин роботи він скинеться в нуль. Проте, завдяки особливостям арифметики беззнакових чисел, обчислення `endTime - startTime` все одно дасть коректний результат, якщо інтервал вимірювання не перевищує ці 70 хвилин.

Завдання

1. Визначте час необхідний для визначення `sum` за допомогою функції `micros()`:

```
float sum = 0;
for (int i = 0; i < 100; i++)
{
    sum += i * (5.5 / 256.0);
}
Serial.println(sum);
```

Оптимізуйте цикл, прибравши зайві обчислення за його межі та повторно перевірте швидкодію.

2. Створіть три фрагменти програми, які додають 100 елементів масивів типів `int`, `int8_t` та `float`. Виміряйте час виконання кожного варіанту. Зробіть висновок про вплив вибору типів на продуктивність.

3. Реалізуйте дві функції, для віднімання двох чисел двома способами — звичайним підходом C++ та асемблерною вставкою. Запустіть обидві функції в циклах, наприклад по 30 000 разів, та виміряйте різницю у часі виконання (використовуйте **volatile** для змінної **results**, щоб компілятор не оптимізував цикл). Поясніть отримані результати.
4. Застосувавши розглянуті методи, оптимізуйте код, що відповідає за обчислення середнього значення:

```
#include <Arduino.h>

int data[100];
float avg = 0;

void setup()
{
    Serial.begin(9600);
    // Ініціалізація масиву
    for (int i = 0; i < 100; i++)
    {
        data[i] = i % 10;
    }
}

void loop()
{
    float sum = 0;
    unsigned long startTime = micros(); // Початок вимірювання часу

    // Обчислення середнього значення
    for (int i = 0; i < 100; i++)
    {
        // Неефективні обчислення
        sum += (float)data[i] * 1.25 + (data[i] / 3.0) - (data[i] * 0.1);
    }
}
```

```

avg = sum / 100.0;

// Час виконання
unsigned long elapsedTime = micros() - startTime;

Serial.print("Average value: ");
Serial.println(avg);

Serial.print("Execution time (microseconds): ");
Serial.println(elapsedTime);

Serial.println("-----");
delay(1000);
}

```

В наведеному вище коді присутні надлишкові операції в циклі, арифметика з плаваючою крапкою (де можна обійтись цілими числами), виконується зайве перетворення типів.

Додаткове завдання 1. Під'єднати світлодіод до одного з портів мікроконтролера. Написати одну асемблерну вставку для реалізації миготіння світлодіодом.

Додаткове завдання 2. Розробити приклад застосування асемблерної вставки, який демонструватиме перевагу в швидкодії над реалізацією C++.

Додаткове завдання 3. Дослідити залежність часу виконання циклів в завданні № 3 при зміні кількості ітерацій. Результати вивести та пояснення вивести в серійний монітор.

Важливо! Бали за додаткові завдання нараховуються за умови вчасного подання звіту.

Порядок виконання роботи

Крок 1. Вимір часу виконання прикладу циклу:

- a. Створіть новий проект у Wokwi для плати розробника Arduino Uno.
- b. Напишіть код для виконання циклу із завдання та перевірте його працездатність.
- c. Виміряйте час виконання циклу за допомогою функції `micros()` та запишіть результат.
- d. Оптимізуйте цикл, перенісши сталий вираз за межі циклу, щоб уникнути зайвих обчислень на кожній ітерації.
- e. Виміряйте час виконання оптимізованого варіанту та порівняйте з початковим. Зробіть висновок.

Крок 2. Вплив вибору типів даних на продуктивність:

- a. Підготуйте три масиви з однаковою кількістю елементів (наприклад, по 100 елементів), але різного типу (наприклад, `byte`, `int` та `float`).
- b. Заповніть їх довільними числами (за допомогою циклу).
- c. Для кожного масиву створіть окремий цикл підсумовування всіх його елементів.
- d. Проаналізуйте результати та зробіть висновок.

Крок 3. Використання асемблерних вставок:

- a. Інтегруйте в скетч дві надані в завданні № 2 функції.
- b. Виконайте кожну з них по 30 000 разів за допомогою відповідного циклу та виміряйте затрачений на це час. Щоб не дати компілятору пропустити виконання одноманітних ітерацій позначте змінну для збереження результатів додавання за допомогою `volatile`, наприклад так:

```
uint32_t startTime, endTime;

// volatile, щоб компілятор не оптимізував цикл
volatile uint8_t res;
```

```
for (int i = 0; i < iterations; i++)  
{  
    res = addCpp(a, b);  
}
```

- с. Проаналізуйте результати, за бажання виконайте додаткове завдання № 3.

Крок 4. Комплексна оптимізація програми:

- а. Виміряйте час необхідний для виконання наданого варіанту програми в завданні № 4.
- б. Оптимізуйте код, застосовуючи перенесення незмінних обчислень за межі циклу, вибір оптимальних типів даних, локалізацію змінних та, при потребі, асемблерні вставки для критичних ділянок.
- с. Порівняйте швидкодію до та після оптимізації та зробіть висновок про ефективність проведених заходів.

Порядок подання звіту

1. Виконайте вхід до курсу на платформі дистанційного навчання «Сікорський».
2. Перейдіть до розділу, що відповідає поточній роботі та відкрийте форму для подання результатів.
3. У полі для текстової відповіді додайте гіперпосилання на збережений проєкт у середовищі Wokwi.

Важливо! Перед відправленням переконайтеся у працездатності проєкту запустивши симуляцію!

4. Збережіть відповідь для внесення проєкту до бази перевірки викладачем.
-

Контрольні запитання

1. Що таке зайві обчислення у циклах і як вони впливають на продуктивність програми?
2. Чому локалізація змінних може підвищити ефективність роботи програми на мікроконтролері?
3. Як вибір типу даних впливає на швидкодію та розмір програми?
4. Яка роль функції `micros()` у вимірюванні часу виконання коду?
5. У яких випадках доцільно застосовувати асемблерні вставки?
6. Для чого застосовується ключове слово **`volatile`**?

ПРАКТИЧНА РОБОТА № 12

ЗАБЕЗПЕЧЕННЯ ЕНЕРГОЕФЕКТИВНОСТІ ТА ВИКОРИСТАННЯ РЕЖИМІВ ЗНИЖЕНОГО ЕНЕРГОСПОЖИВАННЯ

Мета: Освоїти принципи використання вбудованих режимів енергозбереження мікроконтролера ESP32, набути практичних навичок розробки енергоефективного програмного забезпечення із збереженням заданої функціональності пристрою.

План проведення практичного заняття

- Опрацювання теоретичних матеріалів.
 - Покрокове виконання завдань.
 - Подання результатів (звіту) на перевірку.
-

Теоретичні відомості

Енергозбереження – це комплекс заходів і методів, які дозволяють зменшити споживання електроенергії пристроями без втрати їх функціональності. Для сучасних вбудованих систем, особливо тих, що працюють на батареях або акумуляторах, ефективне управління енергією критично важливе.

Основні підходи до зменшення енергоспоживання:

- Оптимізація коду та алгоритмів. Програми можуть бути написані так, щоб CPU виконував обчислення тільки тоді, коли це дійсно необхідно. Наприклад, варто відмовитись від постійного опитування датчиків та використання блокуючих циклів на користь обробки переривань і ефективного використання таймерів.
- Зменшення тактової частоти та напруги живлення. Деякі мікроконтролери дозволяють динамічно зменшувати тактову частоту або напругу живлення, що знижує споживання енергії пропорційно до зниження продуктивності під час виконання простих задач.

- Деактивація незадіяної периферії. Модулі, які не беруть участі у поточних обчисленнях (Wi-Fi, Bluetooth, АЦП, UART), підлягають деактивації шляхом відключення їхнього тактування або живлення.
- Перехід у режими зниженого енергоспоживання (сну) під час простою. Повна зупинка виконання коду та тактування ядра на час очікування зовнішніх подій або сигналів таймера.
- Використання енергоефективної периферії. Вибір сенсорів, дисплеїв та виконавчих модулів із низьким струмом споживання.

Режими зниженого енергоспоживання ATmega328P

Мікроконтролер ATmega328P підтримує шість режимів зниженого енергоспоживання (Sleep Modes), які дозволяють вимикати окремі функціональні вузли чипа [19, 20]. Вибір режиму залежить від переліку периферійних модулів, що мають залишатися активними, та умов виходу мікроконтролера з режиму сну (пробудження). Нижче наведено опис кожного з цих режимів:

- Idle (Бездіяльність) – CPU зупиняється, а периферія (Flash, I/O, ADC, асинхронні таймери) продовжує працювати. Можливі джерела пробудження: зовнішні переривання (INT0, Pin Change), таймери, ADC, WDT. Споживання зменшується незначно, але програма може швидко продовжити роботу.
- ADC Noise Reduction – CPU зупиняється, ADC працює, інші таймери можуть працювати (Asynchronous Timer). Пробудження можливе через ADC, зовнішні переривання або таймери. Використовується, коли потрібно провести точні вимірювання з меншим шумом, одночасно трохи економлячи енергію.
- Power-down (Повний сон) – практично все зупиняється: CPU, Flash, I/O, ADC. Пробудження: тільки зовнішні переривання або WDT. Це найбільш економний режим, споживання мінімальне.

- Power-save – подібний до POWERDOWN. CPU зупиняється, асинхронний таймер (Asynchronous Timer) працює. Пробудження через зовнішні переривання, таймери, WDT. Дозволяє реалізувати періодичні пробудження без зовнішнього сигналу.
- Standby – CPU вимкнено, периферія живе, деякі таймери продовжують відлік, швидке пробудження. Споживання трохи більше, ніж у POWERDOWN, але час виходу з сну дуже короткий.
- Extended Standby – це розширений варіант Standby, де можлива робота асинхронного таймера, що дозволяє виконувати точні відліки часу навіть у стані сну. Споживання енергії трохи більше, ніж у звичайному Standby, але забезпечує кращий контроль таймерів.

Приклад переведення ATmega328P в режим Power-down та пробудження від зовнішнього переривання (в симуляторі може не працювати коректно):

```
#include <Arduino.h>
#include <avr/sleep.h>

// Пін D2 може викликати зовнішнє переривання (INT0)
const int WAKEUP_PIN = 2;

void setup()
{
  Serial.begin(9600);
  pinMode(WAKEUP_PIN, INPUT_PULLUP);

  // Налаштування режиму сну
  set_sleep_mode(SLEEP_MODE_PWR_DOWN);

  // Дозволяємо CPU засинати в різних режимах
  sleep_enable();

  // Приєднання переривання до D2 (INT0)
```

```

// NULL - функція-обробник не викликається, але CPU прокидається
// CHANGE - реагувати на будь-яку зміну на контакті
attachInterrupt(digitalPinToInterrupt(WAKEUP_PIN), NULL, CHANGE);
}

void loop()
{
  Serial.println("Рахуємо до 3х і засинаємо");
  Serial.println("1");
  delay(1000);
  Serial.println("2");
  delay(1000);
  Serial.println("3");
  delay(1000);
  Serial.println("Ардуіно спить. Для пробудження надішліть щось через UART
(RX/D0 має бути з'єднаний з D2)");
  delay(1000);
  // Виклик сну. CPU зупиняється тут.
  sleep_mode();

  // КОД ПРОДОВЖУЄТЬСЯ З ЦІЄЇ ТОЧКИ ТІЛЬКИ ПІСЛЯ ІМПУЛЬСУ НА D2
}

```

На початку плата виводить через Serial повідомлення (короткий відлік), після чого виконується `sleep_mode()`, переводячи CPU у режим заданий режим сну (`SLEEP_MODE_PWR_DOWN`). Пін D2 підключений до спеціального апаратного переривання (`INT0`). Коли на нього подається сигнал (наприклад, натискають кнопку або підключають його до джерела напруги), він «пробуджує» CPU (навіть з глибокого сну) і мікроконтролер продовжує виконання коду.

У глибокому сні UART вимкнений і Arduino не отримує дані через підключений до ПК USB-кабель, тому надсилання тексту через `Serial` не розбудить плату. Фізичне з'єднання D0 (RX) та D2 дозволить пробуджувати плату за допомогою сигналу від ПК (через USB-Serial), оскільки дані що надходитимуть

на RX будуть змінювати стан D2, що викличе пробудження CPU. Інакше кажучи, це «штучний спосіб» пробудження через Serial, коли пряме використання UART неможливе через глибокий сон.

Енергозбереження ESP32

На відміну від ATmega328P, мікроконтролери ESP32 мають спеціалізовані апаратні блоки, які можуть працювати незалежно від основних CPU.

Двома основними режимами збереження енергії в ESP32 є Light-sleep (легкий сон) та Deep-sleep (глибокий сон) [21].

При переході в **Light-sleep** – цифрові периферійні модулі, більшість оперативної пам'яті та ядра CPU переводяться у стан зниженої активності (clock-gated), а їх напруга живлення зменшується. Після виходу з цього режиму периферія, RAM і CPU відновлюють роботу, при цьому всі внутрішні стани зберігаються, тож програми продовжують виконання без втрати даних.

При переході в **Deep-sleep** – повністю вимикаються CPU, більшість RAM і всі цифрові периферійні модулі, що використовують основний тактовий сигнал (APB_CLK). Стан процесора втрачається, а після пробудження ESP32 фактично виконує повторне завантаження програми (reset), хоча дані для відномлення роботи зберігають частини чипа, що залишаються з живленням:

- Контролер годинника реального часу RTC – керує таймерами та сигналами пробудження. Слідкує за подіями на GPIO або від таймерів, щоб вивести ESP32 зі сну.
- Ультранизькоспоживний співпроцесор (ULP) – виконує нескладні обчислення, моніторинг значень АЦП та сенсорів. Може ініціювати пробудження основного CPU при настанні певних умов. Зазвичай ULP програмується на мові асемблера [22].
- RTC FAST memory – швидка пам'ять, що використовується для збереження даних, що будуть негайно необхідні після пробудження, наприклад, критичних змінних або стану програми.

- RTC SLOW memory – повільна пам'ять, що зазвичай використовується для більш великих структур даних.

Пам'ять називається «RTC memory», оскільки вона живиться від блоку RTC, який працює навіть коли основні ядра CPU вимкнені (під час Deep-sleep). Перед переходом у Deep-sleep всі важливі дані, які будуть потрібні після пробудження, потрібно записати в RTC memory, бо звичайна RAM (SRAM) втратить свій вміст після вимкнення живлення. Щоб записати змінну в RTC memory необхідно перед її типом додати спеціальний атрибут (`RTC_DATA_ATTR` або `RTC_NOINIT_ATTR` – для швидкої та повільної пам'яті відповідно), наприклад:

```
RTC_DATA_ATTR int counter = 0;
```

ESP32 підтримує кілька способів пробудження з енергозберігаючих режимів. Найпоширенішим є пробудження за сигналом з GPIO, коли зміна логічного рівня на вхідному піні активує мікроконтролер. Інший варіант – пробудження за таймером RTC, який дозволяє автоматично «розбудити» пристрій через заданий проміжок часу. Крім того, ESP32 може прокидатися за подіями від ULP-співпроцесора та тач-контролера, який вимірює ємність на спеціальних «сенсорних» GPIO-пінах.

Важливо, що у режимах Light-sleep і Deep-sleep обидва бездротові модулі (Wi-Fi та Bluetooth) не працюють. Тому існує ще один проміжний режим – Modem-sleep.

Modem-sleep не є режимом сну в класичному розумінні, оскільки ядро CPU продовжує безперервне виконання програми. Цей режим призначений для оптимізації енергоспоживання модулів Wi-Fi та Bluetooth, які деактивуються за відсутності мережевого трафіку. ESP32 періодично активує Wi-Fi модуль лише на короткі проміжки часу для прийому Beacon-кадрів від точки доступу та перевірки наявності буферизованих даних. Якщо вхідні пакети відсутні, радіомодуль повертається у стан із зниженим енергоспоживанням. Це дозволяє підтримувати

активне мережеве з'єднання та зменшує середній струм споживання на десятки міліампер порівняно з режимом постійної активності.

Основні функції керування режимами сну ESP32 включають [21]:

- `esp_light_sleep_start()` – переведення мікроконтролера в режим Light-sleep;
- `esp_deep_sleep_start()` – переведення в режим Deep-sleep;
- `esp_sleep_enable_timer_wakeup(time_in_us)` – налаштування таймера для пробудження через заданий час (мікросекунди);
- `esp_sleep_enable_ext0_wakeup(GPIO_NUM_X, level)` – пробудження при зміні рівня на одному піні;
- `esp_sleep_enable_ext1_wakeup(mask, mode)` – пробудження при зміні на кількох пінах одночасно;
- `esp_sleep_enable_touchpad_wakeup()` – пробудження дотиком до сенсорного піну (T0–T9);
- `esp_sleep_enable_ulp_wakeup()` – пробудження за подією від ULP-співпроцесора.

Приклад переходу в Light-sleep та пробудження від таймера (через 5 секунд):

```
#include <Arduino.h>
#define timeToSleep 5000000 // 5 секунд в мікросекундах

int counter = 0;

void setup()
{
    Serial.begin(9600);
    // Увімкнення пробудження від таймера
    esp_sleep_enable_timer_wakeup(timeToSleep);
}
```

```

void loop()
{
    Serial.println("Тричі збільшуємо змінну, що зберігається в RAM");

    Serial.println(++counter);
    delay(1000);
    Serial.println(++counter);
    delay(1000);
    Serial.println(++counter);
    delay(1000);

    Serial.println("ESP32 засинає");
    delay(1000);

    esp_light_sleep_start();    // Виклик сну (light-sleep)

    // КОД ПРОДОВЖУЄТЬСЯ З ЦІЄЇ ТОЧКИ ТІЛЬКИ ПІСЛЯ 5-ти СЕКУНД

    Serial.println("ESP32 прокидається");
}

```

Примітка. У симуляторі приклад може працювати некоректно.

Спочатку в прикладі виконується налаштування пробудження за таймером за допомогою функції `esp_sleep_enable_timer_wakeup(timeToSleep)`, після чого ESP32 тричі збільшує змінну `counter`, що зберігається в оперативній пам'яті. Потім виклик `esp_light_sleep_start()` переводить мікроконтролер у режим легкого сну на 5 секунд. Після пробудження виконання коду продовжується з тієї самої точки, а значення змінної зберігається, оскільки RAM під час сну не вимикається повністю.

Приклад переходу в Deep-sleep та пробудження від таймера (через 5 секунд):

```

#include <Arduino.h>
#define timeToSleep 5000000 // 5 секунд в мікросекундах

```

```

int counterRAM = 0;
RTC_DATA_ATTR int counterRTC = 0;
void setup()
{
    Serial.begin(9600);
    esp_sleep_enable_timer_wakeup(timeToSleep); // Увімкнення пробудження
від таймера
}

void loop()
{
    Serial.println("Змінна, що зберігається в RAM: " +
String(counterRAM++));
    delay(1000);
    Serial.println("Змінну, що зберігається в RTC memory: " +
String(counterRTC++));
    delay(1000);
    Serial.println("ESP32 глибоко засинає");
    delay(1000);

    esp_deep_sleep_start(); // Виклик сну (Deep-sleep)

    //ПІСЛЯ ПРОБУДЖЕННЯ МІКРОКОНТРОЛЕР ПОЧНЕ З ПОЧАТКУ
    Serial.println("Сюди не дійде");
}

```

Примітка. У симуляторі приклад може працювати некоректно.

Приклад демонструє втрату даних у оперативній пам'яті та використання спеціалізованої RTC-пам'яті для збереження критичних змінних. На початку вмикається пробудження від таймера кожні 5 секунд (як і в попередньому прикладі). Далі – основний цикл виводить значення двох змінних та збільшує їх на одиницю: **counterRAM** (звичайна змінна, що зберігається в RAM) та **counterRTC**

(змінна, позначена `RTC_DATA_ATTR`). Оскільки режим Deep Sleep повністю вимикає живлення основних компонентів, включно з RAM, після пробудження ESP32 виконує повний апаратний скид, і значення `counterRAM` повертається до початкового 0. Натомість, значення `counterRTC` зберігається у спеціальній енергонезалежній RTC-пам'яті, що дозволяє програмі «пам'ятати» попередній стан і продовжувати підрахунок ітерацій.

Завдання

1. Створити проект для ESP32 та перевірити працездатність прикладів програм з теоретичних відомостей.
2. Підключити датчик температури і реалізувати зчитування його показів після кожного пробудження ESP32. Виводити зчитані значення на серійний монітор.
3. Створити унікальну URL-адресу для тестування за допомогою сайту <https://webhook.site/>. Після кожного пробудження реалізувати відправку значень температури на згенеровану адресу за протоколом HTTP.

Додаткове завдання 1. Налаштувати пробудження від дотику, натискання кнопки або одночасної зміни стану кількох пінів.

Додаткове завдання 2. Запрограмувати ULP для перевірки температури в режимі Deep-sleep та пробудження головного CPU при досягненні порогового значення (напр. 30 °C).

Важливо! Бали за додаткові завдання нараховуються за умови вчасного подання звіту.

Порядок виконання роботи

Крок 1. Створення та перевірка базового проекту:

- a. Відкрийте симулятор Wokwi та створіть новий проект для плати ESP32.
- b. Завантажте приклади коду з теоретичних відомостей та проаналізуйте їх.

Крок 2. Підключення та використання датчика температури:

- a. Підключіть до ESP32 датчик температури (DHT22) та імпортуйте необхідну бібліотеку (напр. **DHT.h**).
- b. Реалізуйте програму, в якій після кожного пробудження з глибокого сну виконується: ініціалізація датчика; зчитування температури; виведення значення температури на серійний монітор. Для періодичного пробудження мікроконтролера, наприклад кожні 10 секунд, використовуйте функцію **esp_sleep_enable_timer_wakeup**.

Крок 3. Передача даних на віддалений сервер:

- a. Відкрийте сайт <https://webhook.site> та згенеруйте новий URL. Збережіть унікальну згенеровану адресу (виглядає як <https://webhook.site/xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxx>), саме на неї ESP32 надсилатиме дані після кожного пробудження.
- b. Для роботи з мережею підключіть бібліотеки **WiFi.h** та **HTTPClient.h**.
- c. Після кожного пробудження ініціалізуйте Wi-Fi, дочекайтеся підключення, зчитайте температуру з датчика і відправте показник на webhook.site за допомогою HTTP-запиту, напр.:

```
String url = "https://webhook.site/ВАША_АДРЕСА?temp=" + String(temperature);  
http.begin(url);  
int code = http.GET();  
http.end();
```

- d. Перевірте працездатність програми. Після переходу ESP32 у Deep-sleep звичайна оперативна пам'ять стирається, тому для збереження даних, які мають залишатися після пробудження використайте пам'ять RTC.
-

Порядок подання звіту

1. Виконайте вхід до курсу на платформі дистанційного навчання «Сікорський».
2. Перейдіть до розділу, що відповідає поточній роботі та відкрийте форму для подання результатів.
3. У полі для текстової відповіді додайте гіперпосилання на збережений проєкт у середовищі Wokwi.

Важливо! Перед відправленням переконайтеся у працездатності проєкту запустивши симуляцію!

4. Збережіть відповідь для внесення проєкту до бази перевірки викладачем.
-

Контрольні запитання

1. Що таке «сон» в контексті мікроконтролерів?
2. Які режими енергозбереження реалізовано в Arduino Uno та ESP32?
3. Які події можуть пробудити мікроконтролери з глибокого сну?
4. Чим режим Deep-sleep відрізняється від Light-sleep?
5. Як поводить ся пам'ять після виходу з Deep-sleep та як не втратити дані?

ПРАКТИЧНА РОБОТА № 13

МАШИННЕ НАВЧАННЯ МОДЕЛІ ДЛЯ КЛАСИФІКАЦІЇ ДИНАМІЧНИХ ПРОЦЕСІВ

Мета: Ознайомитись з методами підготовки даних для створення моделі машинного навчання, а також навчитись створювати та оптимізовувати такі моделі для подальшої реалізації інтелектуальних функцій у програмованих пристроях.

План проведення практичного заняття

- Опрацювання теоретичних матеріалів.
 - Покрокове виконання завдань.
 - Подання результатів (звіту) на перевірку.
-

Теоретичні відомості

TinyML (Tiny Machine Learning) – це напрямок штучного інтелекту, що присвячений розробці та впровадженню моделей машинного навчання для пристроїв з обмеженими ресурсами (мікроконтролерів, сенсорних вузлів, IoT-пристроїв). Основна ідея TinyML – перенести інтелект ближче до сенсорів, тобто виконувати обробку даних на місці (Edge AI), без постійного підключення до Інтернету та хмарних сервісів. TinyML дозволяє: зменшити затримку, адже дані не передаються в мережу; підвищити енергоефективність; забезпечити приватність, бо дані не залишають пристрій; зменшити навантаження на мережу та сервери.

Визначивши, що саме має розпізнавати чи передбачати система (рух, звук, жести, аномалії тощо), можливо перейти до створення та розгортання TinyML-моделі, що включає наступні типові етапи:

- 1) **Підготовка даних** – це, перш за все, збір «сирих» сигналів із сенсорів (акселерометр, мікрофон, камера тощо), додавання міток (класів), а також очищення, нарізання на фрагменти, нормалізація, попереднє перетворення сигналів у більш інформативний вигляд, якщо це потрібно (спектр, статистика, фільтрація).

- 2) **Конфігурація, навчання та оптимізація моделі** включає вибір архітектури, налаштування параметрів, тренування моделі на зібраних даних, квантизацію, зменшення розміру, оптимізацію швидкодії та споживання пам'яті.
- 3) **Оцінювання якості** – перевірка результатів на тестових даних, аналіз точності.
- 4) **Експорт моделі у форматі для мікроконтролера** (C++ бібліотека, TensorFlow Lite Micro модель або Arduino-бібліотека) – навчена модель та вся обчислювальна логіка перетворюються у компактний формат, придатний для виконання на малопотужних мікроконтролерах. Генерується набір файлів, який містить модель, параметри нормалізації, функції для обробки вхідних даних та весь необхідний код для виконання інференції.
- 5) **Розгортання** – це процес інтеграції навченої та оптимізованої моделі у програмне забезпечення мікроконтролера. Для цього, до проєкту (для ESP32/Arduino/тощо) підключається бібліотека з моделлю, а сама інференція, на практиці, викликається як звичайна функція, яка отримує підготовлені дані і повертає прогноз.
- 6) **Валідація та подальше вдосконалення** – перевірка роботи в реальних умовах, збір додаткових даних, повторне навчання моделі.

Підготовка даних

Підготовка даних – це перший і один з найважливіших етапів створення моделі машинного навчання (у тому числі TinyML). На цій стадії формуються вхідні дані, на яких у подальшому буде навчатися модель. Від якості, репрезентативності та чистоти даних залежить точність, стабільність і узагальнювальна здатність моделі.

Для задач TinyML дані зазвичай збирають з сенсорів, підключених до мікроконтролера, на якому надалі працюватиме модель. Це дозволяє врахувати шум, неточності АЦП та реальні умови середовища. Давачі прискорення та гіроскопи (MPU6050, LSM6DS3, ADXL345) – використовуються для виявлення

рухів, жестів, положення тіла тощо. Давачі звуку (мікрофони, MEMS) – для розпізнавання команд, шумів, кроків, ударів; давачі зображення (камери) – для класифікації об’єктів. Давачі температури, вологості, тиску, газів – у завданнях прогнозування, контролю середовища, діагностики станів тощо.

При роботі з часовими рядами (наприклад, дані з акселерометра), «сирі» покази приходять як послідовність чисел у часі: t_0, t_1, t_2 . Модель же не працює з нескінченним потоком – вона потребує обмежений контекст, тобто сегмент сигналу фіксованої довжини. Цей сегмент називають **вікном (Window)**. Наприклад, при розпізнаванні жестів і рухів, форма сигналу у вікні – це джерело ознак, які дозволяють відрізнити один рух від іншого. Вікно дає часові закономірності, оскільки, якщо подавати в модель лише одне вимірювання (за один момент часу), вона нічого не «зрозуміє» про рух – лише про поточне положення. Розмір вікна треба підбирати під тривалість події. Кожен фрагмент запису повинен містити:

- **Мітку (label)** – вказує, що саме відбувається у поточному фрагменті даних, наприклад рух вгору-вниз, вліво-вправо. Якщо вхідними даними є рисунок, то в мітці пояснюється, що на ньому зображено. Без міток модель не навчиться класифікувати.
- **Часові мітки (timestamps)** – фіксують часову послідовність надходження даних при роботі з часовими рядами (розпізнавання жестів, обробка аудіосигналів тощо).
- **Набір числових значень** – виміряні параметри або первинні дані з давачів. Наприклад з акселерометра:

```
accX, accY, accZ
-0.2, 0.98, 0.05
-0.3, 0.95, 0.07
...
```

Для навчання дані зазвичай організовують у вікна фіксованої тривалості/розміру (наприклад, 2000 мс із частотою 100 Гц). Такий фрагмент перетворюється на один приклад для навчання. Фрагменти повинні бути

представлені у цифровому вигляді, найчастіше у форматі CSV, JSON або спеціальному форматі платформи (наприклад, «.ei» у Edge Impulse).

Визначення ідеального обсягу даних та балансу класів для навчання TinyML-моделі є скоріше практичним, емпіричним процесом, ніж математичною формулою. Жорстких формул, що визначають точну кількість прикладів чи хвилин, не існує, оскільки все залежить від складності завдання та якості даних. Для простих задачах (наприклад, розпізнавання жестів) рекомендовано:

- мінімум 5-15 якісних записів для кожного класу. Для надійного результату – 30-100. Важливо, щоб дані були різноманітними (записи від різних людей, з різною швидкістю руху, амплітудою тощо), а не просто великими за обсягом;
- кожен запис повинен бути принаймні такої ж довжини як розмір вікна або довший;
- баланс між класами, тобто кількість записів кожного типу повинна бути приблизно однаковою. Якщо дані нерівномірні, наприклад, певний рух зустрічається частіше, модель може «схилятися» до нього.

Всі зібрані дані слід рівномірно розділити на дві частини: **навчальний набір** (70-80%) – дані, що використовуватимуться для навчання моделі; **тестовий набір** (20-30%) – для остаточної оцінки точності моделі.

Оскільки якість даних безпосередньо впливає на точність та швидкість роботи кінцевої TinyML-моделі – застосовують методи попередньої обробки даних:

- **Сегментація** – сигнали розбиваються на вікна фіксованої довжини. Кожне вікно обробляється як окремий приклад, що подається на вхід моделі.
- **Фільтрація шуму** – видалення випадкових коливань, які не несуть корисної інформації.
- **Видалення артефактів** – ідентифікація та усунення нетипових, помилкових або відсутніх даних (пропущених значень), які можуть спотворити процес навчання моделі.

- **Нормалізація** – масштабування числових значень до єдиного, стандартного діапазону (наприклад, від 0 до 1). Це забезпечує, що жодна ознака не домінує над іншими лише через свою більшу амплітуду.

Приклад збору даних з акселерометра

Основна мета збору даних – отримати репрезентативні вимірювання прискорень по осях X, Y, Z під час виконання різних рухів пристроєм (наприклад: idle; left-right; forward-backward; up-down) для подальшої обробки та навчання моделі розпізнавання жестів. Для збору даних буде використовуватись акселерометр MPU6050 та плата ESP32 в симуляторі Wokwi (рис. 13.1). Дані зберігатимуться у файлах в форматі CSV.

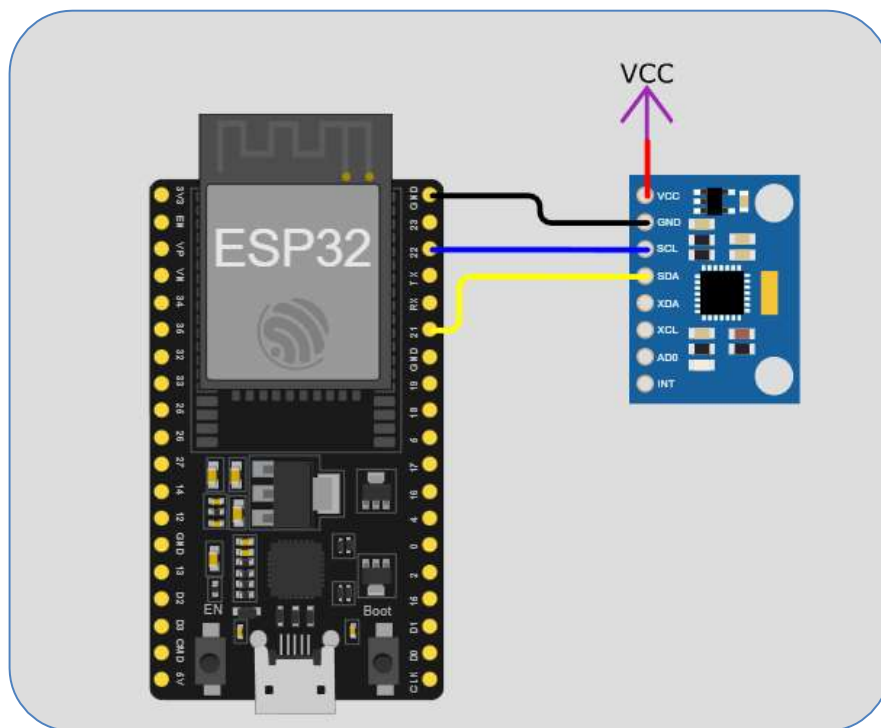


Рис. 13.1. Схема

Програма для збору одного фрагменту даних (розмір вікна – 10 секунд):

```
#include <Arduino.h>
#include <Adafruit_MPU6050.h>

#define FREQUENCY_HZ      100
#define INTERVAL_MS       1000 / FREQUENCY_HZ
```

```

#define NUM_SAMPLES          1000

Adafruit_MPU6050 mpu;

void setup()
{
    Serial.begin(115200);
    mpu.begin();
    mpu.setAccelerometerRange(MPU6050_RANGE_2_G);
    mpu.setFilterBandwidth(MPU6050_BAND_21_HZ);

    Serial.println("Розпочинайте виконувати рух");
    Serial.println("Запис даних розпочнеться через:");
    Serial.println("3.....") ;
    delay(1000);
    Serial.println("2.....") ;
    delay(1000);
    Serial.println("1.....");
    delay(1000);
    Serial.println("GO");

    float acc_x, acc_y;
    unsigned long timestamp;
    unsigned long start_timestamp;
    Serial.println("timestamp,accX,accY");

    start_timestamp = millis();
    for(int i = 0; i< NUM_SAMPLES; i++)
    {
        timestamp = millis();
        sensors_event_t a, g, temp;
        mpu.getEvent(&a, &g, &temp);

        acc_x = a.acceleration.x;

```

```

    acc_y = a.acceleration.y;

    Serial.print(timestamp - start_timestamp);
    Serial.print(",");
    Serial.print(acc_x);
    Serial.print(",");
    Serial.println(acc_y);

    while (millis() < timestamp + INTERVAL_MS);
}

Serial.println("");
Serial.println("Дані зібрано, скопіюйте їх у файл name.number.csv");
Serial.println("Де \"name\" - мітка руху, \"number\" - номер фрагмента
даних");
}

void loop()
{
}

```

На початку наведеного коду підключається бібліотека для роботи з тривісним акселерометром MPU6050, а також створюється відповідний об'єкт «**mpu**». Далі оголошуються параметри:

- **FREQUENCY_HZ** – частота вимірювань у герцах (100 Гц означає, що зчитування відбувається 100 разів на секунду).
- **INTERVAL_MS** – інтервал між двома послідовними вимірюваннями (10 мс).
- **NUM_SAMPLES** – кількість точок (зразків), які потрібно записати під час одного сеансу збору даних (у цьому прикладі – 1000).

Таким чином, під час виконання програма збере 10 секунд даних (1000 зразків $\times$ 10 мс = 10 с).

У конфігураційній функції **setup** ініціалізується послідовний інтерфейс (**Serial**) для виведення зібраних даних у монітор послідовного порту. Далі виконується запуск сенсора та встановлюється діапазон вимірювання прискорення $\pm 2g$, що є достатнім для реєстрації жестів і рухів руки. Для усунення високочастотних завад (вібрацій від двигунів, механічних коливань поверхні та електричного шуму) задіюється вбудований фільтр нижніх частот (ФНЧ). Значення частоти зрізу 21 Гц відповідає апаратному налаштуванню вбудованого ФНЧ акселерометра. Вона достатньо висока, щоб не «зрізати» реальні рухи пристрою і водночас достатньо низька, щоб прибрати дрібні вібрації або електричний шум.

Перед початком запису користувачу дається 3 секунди на підготовку – наприклад, щоб встати у вихідне положення або почати виконувати певний рух.

Цикл **for** виконує 1000 ітерацій, зберігаючи інтервал 10 мс між ними завдяки вбудованій умові в **while**. Під час кожної ітерації зчитується поточне прискорення по двом осям (X,Y) та виводиться на серійний монітор у вигляді рядка CSV (значення, розділені комою). Також, в процесі, визначається час кожного виміру та додається відповідний **timestamp**:

```
timestamp,accX,accY
0,0.1,1.5
10,0.2,1.6
20,3.15,1.8
```

Слід зазначити, що використана бібліотека одразу повертає прискорення давача в m/c^2 замість g .

Після закінчення циклу на екрані з'являється повідомлення, що дані можна скопіювати та зберегти у файл CSV, наприклад:

- leftright.1.csv – перший набір даних для руху «left-right»;
- leftright.2.csv – другий набір даних для руху «left-right»;
- idle.1.csv – перший набір для стану «спокою»;
- idle.2.csv – другий набір для стану «спокою».

Після збору необхідної кількості записів, можна одразу переходити на наступний етап – навчання моделі.

Навчання моделі

На цьому етапі модель «вчиться» розпізнавати закономірності у зібраних сигналах, наприклад, відрізняти, коли пристрій рухається вгору-вниз, вперед-назад або залишається нерухомим.

Для побудови та навчання моделі зручно використовувати онлайн-платформу **Edge Impulse** [23]. Вона створена спеціально для роботи з мікроконтролерами та сенсорами і дозволяє виконати всі необхідні кроки без складного програмування. У межах однієї системи можна завантажити дані, попередньо їх обробити, створити нейронну мережу, навчити її, протестувати точність і зрештою експортувати готову модель у форматі, сумісному з Arduino, ESP32 та іншими мікроконтролерами.

Передусім користувач завантажує зібрані дані (напр. CSV-файли) у систему, де вони автоматично групуються за мітками (з назви файлу) та розподіляються на навчальну та тестову вибірки.

Далі необхідно визначити, які саме **ознаки (features)** модель буде використовувати для розпізнавання. Це надзвичайно важливий етап, оскільки модель не працює безпосередньо з «сирими» даними з акселерометра чи гіроскопа – замість цього вона аналізує математичні характеристики цих сигналів, що допомагають описати їхній стан. Наприклад, система може розраховувати середнє значення, енергію сигналу, частотні компоненти або спектральні коефіцієнти. Такі перетворення допомагають виділити закономірності, які неможливо побачити у звичайному часовому ряді. Формування ознак у Edge Impulse зазвичай виконується автоматично – користувачу достатньо обрати тип блоку обробки (наприклад Spectral features). Після цього система створює набір ознак для кожного фрагмента даних для навчання нейронної мережі. Процес навчання відбувається на серверах Edge Impulse. Користувач може контролювати базові налаштування, наприклад кількість епох, розмір шару або швидкість навчання, проте більшість складних аспектів автоматизовані. Вхідні дані проходить через кілька внутрішніх шарів нейронної мережі, де відбувається навчання розпізнанню патернів руху, і на виході видаються ймовірності для кожного з можливих рухів. Мережа порівнює свої

передбачення з правильними мітками, обчислює помилку й поступово коригує внутрішні ваги, щоб робити точніші передбачення. Процес повторюється багато разів, поки помилка не стане мінімальною, а точність – прийнятно високою.

Важливою перевагою Edge Impulse є можливість **оптимізації моделі**. Сервіс автоматично зменшує кількість параметрів нейронної мережі, використовуючи методи квантування, прямої конверсії у формат TensorFlow Lite та зменшення розрядності ваг. Завдяки цьому модель займає менше пам'яті та споживає менше енергії. У більшості випадків вдається знизити розмір моделі в декілька разів без суттєвої втрати точності.

Результатом цього етапу є навчена модель, яку потім можливо перевірити на тестових даних. Після завершення навчання, платформа показує результати у вигляді графіків і таблиць, за якими можна оцінити, наскільки добре модель розпізнає кожен тип руху. Модель фактично вміє самостійно «розуміти» сигнали від сенсора і приймати рішення без участі людини. Якщо точність задовільна, модель можна експортувати у різних форматах, напр. TensorFlow Lite (.tflite) або як бібліотеку для Arduino IDE. Така модель може бути інтегрована у програму для мікроконтролера, де за допомогою її функцій виконуватиметься класифікація рухів у реальному часі.

Завдання

1. Доопрацювати програму з прикладу, для збору даних за трьома осями прискорення (X, Y, Z). Тривалість фрагменту змінити на 5 секунд.
2. Підготувати навчальні дані. Записати не менше чотирьох класів руху:
 - *idle* – відсутність руху (спокій);
 - *leftright* – рух пристрою вліво-вправо;
 - *forwardbackward* – рух вперед-назад;
 - *updown* – рух вгору-вниз.

Для кожного класу зібрати не менше 5 фрагментів тривалістю по 5 секунд.

3. Створити проект в Edge Impulse, завантажити зібрані дані та виконати навчання нової моделі.
 4. Записати декілька додаткових фрагментів для кожного руху, з різною амплітудою та тривалістю. Завантажити ці файли як тестові дані (Testing data) та перевірити, наскільки конкретно модель класифікує нові зразки. За необхідності – повторити навчання з більшим обсягом даних або зміненими параметрами фільтрації/архітектури.
 5. Експортувати модель у форматі Arduino Library.
-

Порядок виконання роботи

Крок 1. Перевірка працездатності прикладу для збору даних:

- a. В симуляторі Wokwi зібрати схему (рис. 13.1) та додати код прикладу з теоретичних відомостей.
- b. Завантажити бібліотеку для давача MPU6050.
- c. Запустити програму та впевнитись у формуванні фрагменту даних тривалістю 10 секунд у форматі CSV.
- d. За допомогою інтерфейсу Wokwi, спробувати безперервно змінювати показники прискорення під час запису даних. Перевірити відображення виконаних рухів в серійному моніторі.

Крок 2. Доопрацювання програми для збору даних за всіма трьома осями акселерометра:

- a. Додати зчитування прискорення по осі Z, щоб отримувати дані по всіх трьох осях (X, Y, Z).
- b. Змінити формат виводу даних на серійний монітор, щоб вони включали чотири стовбчика – один для **timestamp** та три для прискорення по кожній з осей.
- c. Зменшити тривалість одного фрагменту з 10 секунд до 5 секунд, змінивши **NUM_SAMPLES**.

Крок 3. Підготовка даних у вигляді CSV-файлів:

- a. Створити папку «accdata», а в ній папки «training» та «testing»
- b. В папці «accdata» створити порожні CSV-файли для кожного руху, з іменами:
 - ✓ idle.1.csv, idle.2.csv, idle.3.csv, idle.4.csv, idle.5.csv;
 - ✓ leftright.1.csv, leftright.2.csv, leftright.3.csv, leftright.4.csv, leftright.5.csv;
 - ✓ forwardbackward.1.csv, forwardbackward.2.csv, forwardbackward.3.csv, forwardbackward.4.csv, forwardbackward.5.csv;
 - ✓ updown.1.csv, updown.2.csv, updown.3.csv, updown.4.csv, updown.5.csv.
- c. Запустіть програму. Коли в серійному моніторі розпочнеться відлік – одразу починайте імітувати один з рухів, плавно пересуваючи повзунок однієї з осей прискорення акселерометра в Wokwi (рис. 13.2).

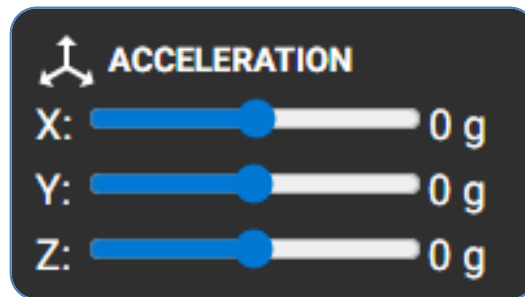


Рис. 13.2. Елементи керування прискоренням пристрою

- ✓ Для імітації руху leftright – на протязі 5-ти секунд, безперервно, пересувайте повзунок для осі X від мінімального значення до максимального і навпаки.
- ✓ Аналогічно, для імітації рухів forwardbackward та updown – на протязі 5-ти секунд, пересувайте повзунок для осей Y та Z відповідно.
- ✓ Для імітації простою (idle) – жодних повзунків не слід рухати під час запису даних.

Примітка. Під час реєстрації реальних (зокрема складних просторових) рухів за допомогою акселерометра значення прискорення за всіма трьома осями (X, Y, Z) змінюватимуться одночасно.

- d. Запускати програму 25 разів, кожного разу записуючи певний рух на протязі 5-ти секунд. Після кожного виконання програми переносити зібрані дані з серійного монітору в відповідний файл (в Windows редагувати CSV-файли можливо за допомогою «Блокноту», рис. 13.3).

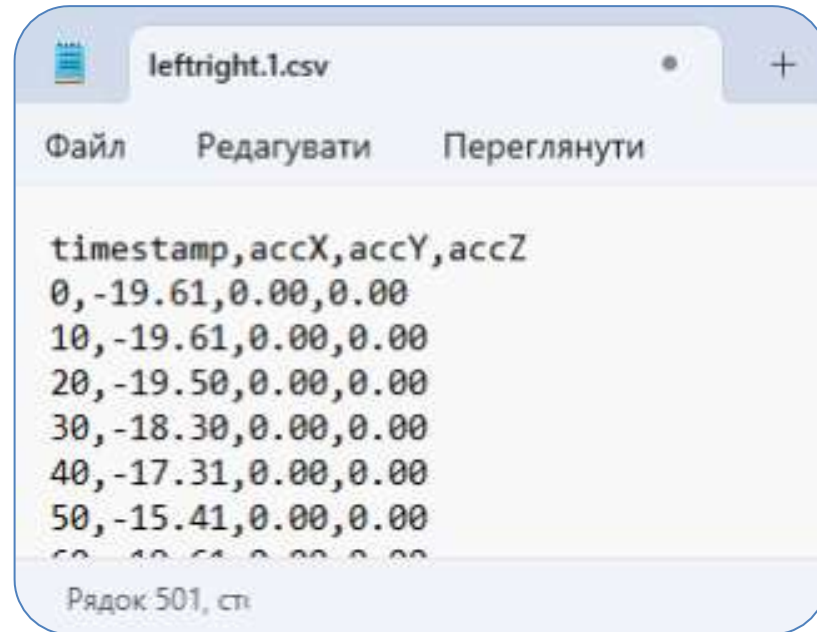


Рис. 13.3. Приклад правильної назви та наповнення CSV-файлу

- e. Розподілити отримані файли між папками «training» (по 4 файли для кожного руху) та «testing» (по 1 файлу для кожного руху). Edge Impulse рекомендує приблизне співвідношення 80%/20%.

Крок 4. Створення проекту в Edge Impulse [24] та вивантаження зібраних даних:

- Зареєструйтесь та увійдіть до студії Edge Impulse:
<https://studio.edgeimpulse.com/>.
- Створіть новий проект або продовжуйте працювати в існуючому.
- Для завантаження зібраних даних оберіть «Data acquisition» з бокового меню.
- Натисніть кнопку «Add data» та оберіть «Upload data».
- Оберіть «Select a folder».
- Натисніть «Вибрати файли» та вкажіть шлях до папки з зібраними даними «acccdata» (рис. 13.4).

Upload mode

☐ Select individual files ?

☒ Select a folder ?

Select files

Вибрати файли файлів: 16

Upload into category

i Upload category will be derived from the structure of your dataset (e.g. samples in a 'train' directory will be uploaded into training data).

Label

☒ Infer from filename ?

Рис. 13.4. Загальна структура інтелектуального пристрою

g. Натисніть «Upload data».

h. Після вивантаження всіх файлів можливо переглянути графічне представлення сирих даних. Переконайтесь, що система правильно розподілила файли між каталогами Training та Test (рис. 13.5).



Рис. 13.5. Приклад відображення файлів в Edge Impulse

Крок 5. Створення імпульсу:

- a. Натисніть «Create impulse» у боковому меню.
- b. В вікні «Time series data» ознайомтесь з поясненнями та встановіть Window size = 2000ms, Window increase = 100 ms, Frequency = 100 Hz.
- c. Натисніть «Add a processing block», оберіть «Spectral Analysis» та натисніть «Add».
- d. Натисніть «Add a learning block», оберіть «Classification» та натисніть «Add».
- e. Натисніть кнопку «Save Impulse» (рис. 13.6).

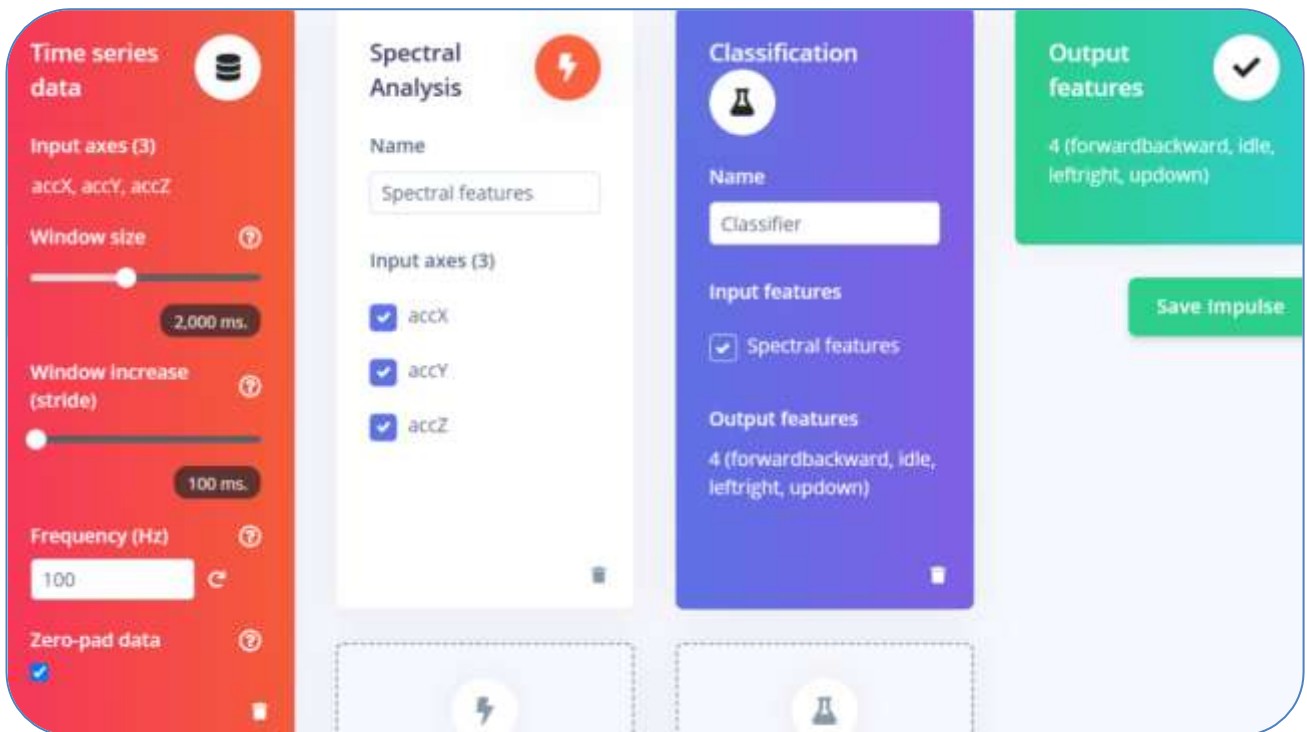


Рис. 13.6. Вікно створення імпульсу

- f. Оберіть «Spectral features» в боковому меню.
- g. Залиште встановлені параметри за замовчування та натисніть «Save parameters» (в майбутньому, якщо виникнуть проблеми з класифікацією, можна спробувати повторно натренувати модель з іншими параметрами).
- h. Натисніть «Generate features» та дочекайтесь завершення процесу.
- i. Оберіть «Classifier» в боковому меню.
- j. Встановіть кількість тренувальних циклів = 100 та натисніть «Save & train».

- k. Перегляньте результати навчання, якщо точність (Accuracy) близька до 100% навчання можна завершити.

Крок 6. Перевірка класифікації рухів:

- a. За допомогою ESP32 запишіть ще декілька додаткових фрагментів для кожного руху, з різною амплітудою та тривалістю. Завантажте ці файли як тестові дані (Testing data) в систему. Для таких зразків можна відключити автоматичне проставлення міток при завантаженні.
- b. Оберіть «Live classification» в боковому меню.
- c. В вікні «Classify existing test sample» оберіть новий фрагмент та натисніть «Load sample».
- d. Дочекайтесь результатів та перевірте чи правильно створена модель визначила рух, а також перегляньте з якою імовірністю вона впевнена в цьому.
- e. Оберіть «Model testing» в боковому меню та натисніть «Classify all», для повторної класифікації всіх тестових даних.
- f. За потреби (при точності <75%) – повторіть навчання з більшим обсягом даних або зміненими параметрами фільтрації/архітектури.

Крок 7. Оптимізація та завантаження навченої моделі:

- a. Оберіть «Deployment» в боковому меню.
- b. Натисніть «Search deployment options» та оберіть «Arduino library».
- c. У розділі «Model optimizations» рекомендується активувати EON Compiler, він оптимізує споживання оперативної пам'яті та енергії моделлю та забезпечить швидше виконання, ніж звичайна реалізація TensorFlow Lite. Також доцільно увімкнути Quantized (Int8) варіант моделі.
- d. В наведеній таблиці можливо перевірити орієнтовні показники споживання пам'яті моделлю, а також орієнтовний час класифікації одного фрагменту (мс).
- e. Під таблицею натисніть «Change target» та оберіть «Espressif ESP-EYE (ESP32 240MHz)».

- f. Натисніть «Build».
 - g. Збережіть завантажений zip-архів (бібліотеку) до наступного заняття, присвяченого інтеграції моделі в програму та виконанню класифікації сигналів в реальному часі безпосередньо на ESP32.
-

Порядок подання звіту

1. Виконайте вхід до курсу на платформі дистанційного навчання «Сікорський».
2. Перейдіть до розділу, що відповідає поточній роботі та відкрийте форму для подання результатів.
3. У полі для відповіді додайте завантажте:
 - ZIP-архів з усіма CSV-файлами;
 - ZIP-архів (Arduino Library) з навченою моделлю;
 - посилання на створений проект в симуляторі Wokwi з виконаними завданнями;
 - скриншот з результатами тестування моделі в Edge Impulse (вікно «Model testing»).

Важливо! Перед відправленням переконайтеся у працездатності проекту запустивши симуляцію!

4. Збережіть відповідь для внесення проекту до бази перевірки викладачем.
-

Контрольні запитання

1. Яке основне призначення етапу збору даних у процесі побудови моделі машинного навчання?
2. Що таке фрагмент у контексті збору даних акселерометра?
3. Чому важливо збирати однакову кількість зразків для кожного класу руху?
4. Яке співвідношення між кількістю зібраних фрагментів і точністю роботи моделі?

5. Яке призначення мають мітки (labels) при зборі фрагментів і чому вони критично важливі для подальшого навчання?
6. Які типи даних використовуються для навчання моделі у цій роботі?
7. Як формується навчальна вибірка для розпізнавання рухів за осями X, Y, Z?
8. Яким чином можна змінити тривалість вибірки (фрагмента) під час збору даних?
9. Для чого створюються CSV-файли і як вони наповнюються?
10. Яке призначення має платформа Edge Impulse, і які основні етапи роботи з нею?
11. Які етапи передбачає створення імпульсу (Impulse) у середовищі Edge Impulse?
12. Яке призначення EON Compiler, чим він відрізняється від стандартного?
13. Які показники характеризують ефективність створеної моделі на мікроконтролері?
14. Які дії потрібно виконати для оптимізації моделі перед її розгортанням на пристрої?

ПРАКТИЧНА РОБОТА № 14

РОЗГОРТАННЯ МОДЕЛІ ШТУЧНОГО ІНТЕЛЕКТУ НА МІКРОКОНТРОЛЕРІ

Мета: Навчитись розгортати модель ШІ на мікроконтролері та здійснити перевірку її роботи в реальному часі.

План проведення практичного заняття

- Опрацювання теоретичних матеріалів.
 - Покрокове виконання завдань.
 - Подання результатів (звіту) на перевірку.
-

Теоретичні відомості

Розгортання (Deployment) – це фінальний і критично важливий етап, який перетворює навчену модель ШІ на виконуваний код у складі мікроконтролера. На цьому етапі модель конвертується у виконуваний код для автономного виконання інференції і завантажується у Flash-пам'ять пристрою у складі єдиної прошивки.

Інференція (Inference) – це процес застосування навченої моделі до нових, невідомих даних для отримання результату (наприклад, числового значення або класифікації). Під час цього модель не змінюється, а лише швидко проганяє вхідні дані через математичні залежності, які були встановлені під час навчання. Наприклад, отримавши на вході звук, записаний мікрофоном, після виконання розрахунків, модель видає прогноз «я чую українську мову». Інференція – це як застосування знань експертом, який більше не вчиться, а просто використовує свої знання для розв'язання нових задач.

Етап розгортання моделі на мікроконтролері включає:

- 1) **Інтеграцію моделі (бібліотеки) в проєкт.** Модель у форматі сумісному з платформою мікроконтролера (наприклад, C++ бібліотека, TensorFlow Lite Micro або Arduino-бібліотека) додається до програмного проєкту як зовнішній модуль. Забезпечується ініціалізація системи перед

використанням моделі, увімкнення необхідних периферійних модулів, перевірка достатності ресурсів тощо.

- 2) **Забезпечення надходження даних від сенсорів та інтерфейсів в модель у правильному форматі та належному обсязі.** Невідповідність вхідного потоку очікуваній структурі або його недостатня точність унеможливають коректне функціонування моделі.

Отримання показників із сенсорів зазвичай відбувається через комунікаційні інтерфейси (наприклад, I2C, SPI, UART). Дані повинні надходити з тією частотою, яку передбачено під час навчання моделі, оскільки модель очікує саме такі інтервали між вимірюваннями. Якщо модель навчалася на фрагментах тривалістю 1 секунда з частотою дискретизації 100 Гц, то під час роботи на пристрої слід забезпечити аналогічні умови.

Крім того, дані мають бути правильно структуровані у вхідному буфері моделі – у відповідній послідовності та кількості. Наприклад, якщо модель очікує вектор із 500 значень, необхідно реалізувати механізм буферизації (накопичення) відліків до досягнення заданого розміру перед запуском інференції. Для приведення даних від давачів до очікуваного моделлю формату слід використовувати константи, згенеровані під час експорту бібліотеки (напр. `FREQUENCY_HZ`, `EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE`, `EI_CLASSIFIER_LABEL_COUNT`), що повністю відповідають параметрам навчання моделі: частоті збору даних, тривалості вхідного вікна даних, кількості класів, розміру вхідного сигналу тощо. Крім цього, первинні дані можуть потребувати нормалізації, масштабування, зміщення або обчислення попередньо визначених ознак.

- 3) **Забезпечення роботи моделі в реальному часі.** Програма повинна реалізовувати регулярний або подієво-орієнтований запуск інференції, під час якого модель отримує актуальний масив даних, здійснює обчислення та повертає результат класифікації, що використовується пристроєм для прийняття рішень або виконання певних дій.

Розробник впроваджує виклик моделі у головний цикл програми або в окреме завдання, якщо використовується операційна система реального часу (RTOS). У більшості застосувань TinyML важливо забезпечити чіткий цикл роботи, який повторюватиметься з визначеною частотою: *збір даних → підготовка → інференція (виконання моделі) → інтерпретація результатів*.

Наприклад, якщо модель розпізнає рухи за допомогою акселерометра, інференція може виконуватись в певні проміжки часу або після накопичення необхідної кількості вимірювань. Важливо, щоб модель виконувалась достатньо швидко, аби не виникало затримок і система адекватно працювала в реальному часі. Після отримання результатів (наприклад, ймовірностей належності руху до певних класів або значення прогнозованого параметра), їх необхідно опрацювати на рівні логіки програми. У випадку класифікації – визначається який клас є найбільш імовірним і це рішення передається на виконавчий рівень, наприклад, для керування двигуном чи генерації звукового сповіщення. Також, слід передбачити обробку виняткових ситуацій, наприклад: якщо модель не впевнена у рішенні; якщо дані неповні або втрачені; якщо процес інференції виконується довше, ніж очікувалось.

- 4) **Фінальне прошивання пристрою та тестування в умовах реального використання.** Важливо перевірити, як система працює на практиці, адже під час симуляції або лабораторного тестування модель може демонструвати хороші результати, але при роботі з «живими» сенсорами та в динамічному середовищі може поводитись непередбачувано.

Окрему увагу приділяють оцінці продуктивності – наскільки швидко модель виконує інференцію і чи встигає це робити в межах заданих часових інтервалів. Якщо модель починає неправильно класифікувати рухи або повільно реагує на зміни – потребується додаткове налаштування або донавчання моделі. У деяких випадках може допомогти коригування частоти виконання моделі або параметрів сенсорів.

Завдання

1. Інтегрувати отриману на попередній роботі бібліотеку наченої моделі III (у форматі Arduino Library) в проект для ESP32.
2. Реалізувати безперервний збір даних з акселерометра MPU6050, формуючи повне вікно даних відповідно до параметрів моделі. Забезпечити передачу цього вікна до функції бібліотеки для здійснення інференції (прогнозування) в реальному часі.
3. Налаштувати виведення на серійний монітор результатів класифікації (прогнозована назва класу та рівень впевненості) разом із вимірним часом інференції.
4. Підключити до ESP32 чотири світлодіоди різних кольорів, за допомогою яких сигналізувати про найбільш імовірний рух, що відбувається в реальному часі.

Порядок виконання роботи

Крок 1. Інтеграція навченої моделі:

- а. Переконайтеся, що у вас є ZIP-архів моделі, експортований з Edge Impulse (назва архіву зазвичай має формат *username-projectname-arduino-x.x.x.zip*).
- б. Відкрийте папку «lib» (рис. 14.1) проекту PlatformIO для ESP32 та перемістіть у неї ZIP-архів.

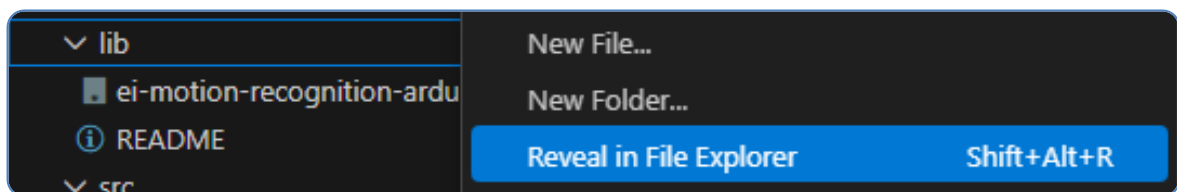


Рис. 14.1. Папка для розміщення архіву з бібліотекою

- в. Відкрийте файл `platformio.ini`. У секції **lib_deps** підключіть усі необхідні бібліотеки, які використовуються моделлю та апаратним забезпеченням:

```
lib_deps = adafruit/Adafruit MPU6050@^2.2.6
```

```
./lib/ei-motion-recognition-arduino-1.0.1.zip
```

- d. Відкрийте головний файл програми **main.cpp**. На початку коду додайте необхідні заголовки:

```
#include <Arduino.h>
#include <Adafruit_MPU6050.h>
#include <Motion_recognition_inferencing.h>
```

Файл *Motion_recognition_inferencing.h* (де *Motion_recognition* – це назва вашого проєкту в Edge Impulse) є головним інтерфейсом між кодом та моделлю ШІ. Він містить усі прототипи функцій, які потрібні для запуску моделі, а також вбудовані константи, які встановлювались під час проєктування Імпульсу в Edge Impulse (розмір буфера, частота, список класів тощо).

Крок 2. Ініціалізація апаратного забезпечення та параметрів:

- a. Визначте інтервал часу в мілісекундах, що необхідний між послідовними опитуваннями датчика, для досягнення встановленої частоти:

```
#define FREQUENCY_HZ      100
#define INTERVAL_MS       (1000 / (FREQUENCY_HZ + 1))
```

- b. Оголосіть об'єкт **mpu** класу **Adafruit_MPU6050**, щоб використовувати методи бібліотеки для ініціалізації датчика та зчитування прискорення:

```
Adafruit_MPU6050 mpu;
```

- c. Підготуйте основний буфер для зберігання зібраних з акселерометра даних:

```
float features[EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE];
size_t feature_ix = 0;
```

Буфер має бути типу **float**, оскільки модель вимагає даних із рухомою комою. **EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE** – це константа, визначена у заголовочному файлі Edge Impulse, і вона містить загальну кількість точок (X, Y, Z), які модель очікує на вході. Індекс (лічильник) **feature_ix** використовується для відстеження поточної позиції запису у буфері **features**.

Коли цей індекс досягає значення `EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE`, це сигналізує, що одне повне вікно даних зібрано і готове для виконання інференції.

- d. Створіть змінну `last_interval_ms`, що використовуватиметься для зберігання часу останнього успішного зчитування та реалізації неблокуючої затримки в основному циклі програми, щоб гарантувати, що наступне зчитування відбудеться лише після того, як мине `INTERVAL_MS`:

```
static unsigned long last_interval_ms = 0;
```

- e. У функції `setup()` ініціалізуйте послідовний інтерфейс (`Serial`) та давач MPU6050. Реалізуйте блокуючий нескінченний цикл, який зупиняє подальше виконання програми за відсутності зв'язку з датчиком. Опціонально можна налаштувати діапазон вимірювань акселерометра відповідно до параметрів, на яких навчалася модель (наприклад, $\pm 2g$):

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

    if (!mpu.begin())
    {
        Serial.println("Не вдалося знайти MPU6050");
        while (1);
    }

    mpu.setAccelerometerRange(MPU6050_RANGE_2_G);
}
```

Крок 3. Реалізація основного циклу програми для безперервного збору даних з акселерометра та здійснення інференції в реальному часі:

```
void loop()
{
    // Оголошення структури для отримання даних з MPU6050
    sensors_event_t a, g, temp;

    // Контроль частоти опитування
    // Перевірка, чи минув необхідний інтервал часу (INTERVAL_MS)
    if (millis() > last_interval_ms + INTERVAL_MS)
    {
        // Оновлення часу останнього зчитування
        last_interval_ms = millis();

        // Зчитування даних
        mpu.getEvent(&a, &g, &temp);

        // Збираються сирі дані
        // Буфер 'features' заповнюється даними про прискорення по X, Y, Z
        features[feature_ix++] = a.acceleration.x;
        features[feature_ix++] = a.acceleration.y;
        features[feature_ix++] = a.acceleration.z;

        // Перевірка заповнення вікна
        // Якщо індекс досяг загального розміру фрейму моделі...
        if (feature_ix == EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE)
        {

            Serial.println("--- Запущено Інференцію (Прогнозування) ---");

            signal_t signal;           // Структура для вхідного сигналу
            ei_impulse_result_t result; // Структура для результатів прогнозу
```

```

// Формування сигналу з буфера 'features' для обробки моделлю
numpy::signal_from_buffer(features,
                           EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE,
                           &signal);

// Запуск класифікатора ШІ
// Модель аналізує сигнал та записує результат у 'result'
run_classifier(&signal, &result, false);

// Вивід результатів та метрик:

// - вивід загального часу, витраченого на прогнозування
Serial.printf("Час Інференції: %d мс.\r\n\r\n",
              (int)result.timing.classification
              + (int)result.timing.dsp);

// - вивід прогнозу (класу з найбільшою впевненістю)
for (size_t ix = 0; ix < EI_CLASSIFIER_LABEL_COUNT; ix++)
{
    Serial.printf("Клас %s: %.2f%%\r\n",
                  result.classification[ix].label,
                  result.classification[ix].value * 100.0);
    // Множення на 100 виконується для виводу у %
}
// Скидання індексу для початку збору нового вікна даних
feature_ix = 0;
Serial.println("-----");
}
}
}

```

Детальніше ознайомитись з використаними функціями та прикладами коду можливо в офіційній документації Edge Impulse [\[25\]](#).

Крок 4. Перевірка розпізнавання рухів:

- a. Скомпілюйте фінальний код та завантажте у мікроконтролер ESP32.
- b. Відкрийте Serial Monitor (Terminal Wokwi) для перегляду виводу програми.
- c. По черзі виконуйте кожен із навчених жестів, рухаючи повзунки давача MPU6050 у симуляторі.
- d. Спостерігайте за реакцією системи після збору кожного повного вікна даних. Проаналізуйте, чи відповідає клас з найвищим рівнем впевненості фактично виконаному жесту. Проаналізуйте продуктивність розгорнутої моделі ШІ на основі виміряного часу інференції.
- e. Приклад виводу результатів в термінал при виконанні різних рухів:

```
--- Запущено Інференцію (Прогнозування) ---  
Час Інференції: 44 мс.  
Клас forwardbackward: 0.78%  
Клас idle: 98.44%  
Клас leftright: 0.00%  
Клас updown: 0.39%  
-----  
--- Запущено Інференцію (Прогнозування) ---  
Час Інференції: 45 мс.  
Клас forwardbackward: 2.73%  
Клас idle: 0.39%  
Клас leftright: 89.06%  
Клас updown: 8.20%  
-----  
--- Запущено Інференцію (Прогнозування) ---  
Час Інференції: 45 мс.  
Клас forwardbackward: 95.70%  
Клас idle: 3.91%  
Клас leftright: 0.39%  
Клас updown: 0.00%  
-----
```

```

--- Запущено Інференцію (Прогнозування) ---
Час Інференції: 45 мс.
Клас forwardbackward: 0.39%
Клас idle: 0.39%
Клас leftright: 2.73%
Клас updown: 96.88%
-----

```

Крок 5. Візуалізація фінального рішення за допомогою світлодіодів:

- a. Підключіть чотири світлодіоди різних кольорів до вільних цифрових виводів GPIO плати ESP32. Для захисту мікроконтролера обов'язково використовуйте обмежувальні резистори (наприклад, 220 Ом).
- b. У коді оголошіть номери відповідних виводів GPIO. Наприклад: **#define LED_IDLE 2** і т.д.
- c. Реалізуйте алгоритм визначення класу з найвищим показником впевненості (максимальним значенням **result.classification[ix].value**). Алгоритм повинен вмикати світлодіод лише визначеного класу, попередньо вимикаючи всі інші.

Порядок подання звіту

1. Виконайте вхід до курсу на платформі дистанційного навчання «Сікорський».
2. Перейдіть до розділу, що відповідає поточній роботі та відкрийте форму для подання результатів.
3. У полі для відповіді завантажте:
 - ZIP-архів (Arduino Library) з моделлю;
 - посилання на створений проект в симуляторі Wokwi з виконаними завданнями.

Важливо! Перед відправленням переконайтеся у працездатності проекту запустивши симуляцію!

4. Збережіть відповідь для внесення проєкту до бази перевірки викладачем.
-

Контрольні запитання

1. Що таке TinyML і чому моделі спеціально оптимізують для мікроконтролерів?
2. Яким чином частота збору даних впливає на роботу моделі?
3. Від чого залежить кількість значень у буфері features?
4. Що таке «вікно даних» (window)?
5. Яким чином тривалість вікна впливає на виконання моделі?
6. Чому важливо забезпечити відповідність частоти вимірювань, тривалості вікон та кількості ознак між етапами навчання та розгортання моделі?
7. Які основні ресурси мікроконтролера можуть обмежувати роботу моделі?
8. Які чинники можуть спричинити зниження точності класифікації моделі під час її тестування на реальному пристрої?

ПЕРЕЛІК ПОСИЛАНЬ

1. Нікітчук А. В. Програмування вбудованих систем Інтернету речей. Конспект лекцій. *КПІ ім. Ігоря Сікорського*. URL: <https://ela.kpi.ua/handle/123456789/71804> (дата звернення: 4.04.2026).
2. Нікітчук А. В. Програмування вбудованих систем Інтернету речей. Лабораторний практикум. *КПІ ім. Ігоря Сікорського*. URL: <https://ela.kpi.ua/handle/123456789/71805> (дата звернення: 4.04.2026).
3. Supported Hardware. *Wokwi Docs*. URL: <https://docs.wokwi.com/getting-started/supported-hardware> (дата звернення: 4.04.2026).
4. Visual Studio Code. URL: <https://code.visualstudio.com> (дата звернення: 4.04.2026).
5. Language Reference. *Arduino Documentation*. URL: <https://docs.arduino.cc/language-reference/#functions> (дата звернення: 4.04.2026).
6. Wokwi-7segment Reference. *Wokwi Docs*. URL: <https://docs.wokwi.com/parts/wokwi-7segment#simulator-examples> (дата звернення: 4.04.2026).
7. Wokwi-dht22 Reference. *Wokwi Docs*. URL: <https://docs.wokwi.com/parts/wokwi-dht22> (дата звернення: 4.04.2026).
8. RX/TX Pin Examples. *Arduino Documentation*. URL: <https://docs.arduino.cc/learn/communication/uart/#rxtx-pin-examples> (дата звернення: 4.04.2026).
9. MPU-6000 and MPU-6050 Register Map and Descriptions. *InvenSense*. URL: <https://invensense.tdk.com/download-resource/rm-mpu-6000a-00-mpu-6000-and-mpu-6050-register-map-and-descriptions> (дата звернення: 4.04.2026).
10. NTPClient. *GitHub*. URL: <https://github.com/arduino-libraries/NTPClient> (дата звернення: 4.04.2026).
11. LiquidCrystal_I2C. *GitHub*. URL: https://github.com/johnrickman/LiquidCrystal_I2C (дата звернення: 4.04.2026).
12. Standard ECMA-404 The JSON data interchange syntax. *Ecma International*. URL: <https://ecma-international.org/publications-and-standards/standards/ecma-404/> (дата звернення: 4.04.2026).
13. Documentation. *ArduinoJson*. URL: <https://arduinojson.org/v7/> (дата звернення: 21.05.2026).

14. Weather Forecast API. *Open-Meteo*. URL: <https://open-meteo.com/en/docs> (дата звернення: 4.04.2026).
15. Comparison of real-time operating systems. *Wikipedia*. URL: https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems (дата звернення: 4.04.2026).
16. The FreeRTOS™ Reference Manual. *Amazon Web Services*. URL: https://www.freertos.org/media/2018/FreeRTOS_Reference_Manual_V10.0.0.pdf (дата звернення: 4.04.2026).
17. FreeRTOS Overview. *Espressif Systems*. URL: <https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/freertos.html> (дата звернення: 4.04.2026).
18. AVR® Instruction Set Manual. *Microchip*. URL: <https://ww1.microchip.com/downloads/en/devicedoc/AVR-Instruction-Set-Manual-DS40002198A.pdf> (дата звернення: 4.04.2026).
19. The Arduino Guide to Low Power Design. *Arduino Documentation*. URL: <https://docs.arduino.cc/learn/electronics/low-power/> (дата звернення: 4.04.2026).
20. ATmega328P Datasheet. *Microchip*. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf (дата звернення: 4.04.2026).
21. Sleep Modes – ESP32. *Espressif Systems*. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/sleep_modes.html (дата звернення: 4.04.2026).
22. ULP Coprocessor programming – ESP32. *Espressif Systems*. URL: <https://docs.espressif.com/projects/esp-idf/en/v4.3/esp32/api-guides/ulp.html> (дата звернення: 4.04.2026).
23. Edge Impulse – The Leading Edge AI Platform. URL: <https://edgeimpulse.com/> (дата звернення: 4.04.2026).
24. Profile – Projects. *Edge Impulse*. URL: <https://studio.edgeimpulse.com/> (дата звернення: 4.04.2026).
25. Data forwarder. *Edge Impulse*. URL: <https://docs.edgeimpulse.com/tools/clis/edge-impulse-cli/data-forwarder> (дата звернення: 4.04.2026).