Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

А. В. Нікітчук

ПРОГРАМУВАННЯ ВБУДОВАНИХ СИСТЕМ ІНТЕРНЕТУ РЕЧЕЙ

Конспект лекцій

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньою програмою «Інтелектуальні технології радіоелектронної техніки»
спеціальності 172 Електронні комунікації та радіотехніка

Електронне мережеве навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2024

УДК 004.4’23.031.6:004.77(07)
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 № 1 від 26.09.2024 р.)
за поданням вченої ради радіотехнічного факультету
(протокол № 6/2024 від 28.06.2024 р.)

Нікітчук А. В.

Н 62 Програмування вбудованих систем Інтернету речей [Електронний ресурс] : конспект лекцій : навч. посіб. для здобувачів ступеня бакалавра за освіт. програмою «Інтелектуальні технології радіоелектронної техніки» спец. 172 Електронні комунікації та радіотехніка / А. В. Нікітчук ; КПІ ім. Ігоря Сікорського. – Електрон. текст. дані (1 файл). – Київ : КПІ ім. Ігоря Сікорського, 2024. – 120 с.

Конспект лекцій допоможе читачам розібратися в специфіці вбудованих систем та методів їх програмування. Увагу приділено системам Інтернету речей, наведено приклади використання популярних комунікаційних протоколів. Розглянуто можливості паралельного виконання багатьох задач з використанням операційних систем реального часу. Лаконічний виклад матеріалу дозволяє швидко засвоїти основні концепції та застосувати їх на практиці. Видання призначене для студентів технічних спеціальностей, а також інженерів зацікавлених у розширенні знань в сфері вбудованих систем та Інтернету речей.

УДК 004.4’23.031.6:004.77(07)

Реєстр. № НП 24/25-048. Обсяг 10,17 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, 2024
© КПІ ім. Ігоря Сікорського, 2024

ЗМІСТ

ВСТУП

Вбудовані системи – це спеціалізовані обчислювальні пристрої, спроектовані для виконання певних завдань або функцій. Їх особливість заключається у вузькому спрямуванні на виконання конкретного завдання та оптимізації під нього. Такі системи зустрічаються в різних сферах життя, від домашньої техніки та автомобілів до медичних приладів та промислових установок.

Вбудовані системи пов'язані із зародженням електронної обчислювальної техніки. Першими вбудованими системами можна вважати електромеханічні системи управління, а однією з перших, впізнаваних в сучасному розумінні вбудованих систем, прийнято вважати навігаційний комп'ютер Apollo, який був встановлений на борту командного та місячного модуля відповідної космічної місії. З появою інтегральних мікросхем та мікроконтролерів вбудовані системи стали більш доступними. Мікроконтролери об'єднали обчислювальний блок, пам'ять і периферійні пристрої на одному чіпі, що зробило їх ідеальними для великої кількості застосувань, від побутової техніки до промислових систем.

Для чого вивчати вбудовані системи?

Вони відіграють ключову роль у впровадженні новітніх технологій, таких як Інтернет речей (IoT), автономні автомобілі, розумна медична техніка та багатьох інших областях. Вивчення вбудованих систем дозволяє студентам та фахівцям розвивати комплексний підхід до роботи з технікою – від фізичного поєднання різноманітних давачів, актуаторів та інших апаратних модулів, до розробки та оптимізації програмного коду для ефективного виконання конкретного завдання. Набуті знання будуть основою для активного участі в інноваційному прогресі та вирішенні технічних викликів.

Сучасні тенденції вбудованих систем:

- **Автоматизація виробництва.** Відіграють ключову роль в покращенні ефективності, точності та безпеки виробничих процесів.
- **Інтернет речей (IoT).** Взаємодія з мережею та обмін даними через Інтернет дозволяє збільшити функціональність пристроїв та забезпечити зручний дистанційний доступ.
- **Хмарні технології.** Співпраця вбудованих систем із хмарними сервісами дозволяє збільшити обсяг збереження даних, використовувати переваги розподіленої обробки даних, динамічної зміни потужності обчислювальних серверів та багато інших корисних послуг, що надаються хмарними провайдерами.
- **Штучний інтелект та машинне навчання.** Забезпечити аналітику великих об'ємів даних, підтримку прийняття рішень та автоматизацію процесів дозволяє впровадження у вбудовані системи так званого «штучного інтелекту».
- **Кібер- та фізична бепека**. Все більше уваги приділяється забезпеченню безпеки. Це стосується як використання вбудованих систем, для підвищення фізичної безпеки, так і забезпечення їх захисту від кіберзагроз.

Сучасні вбудовані системи не лише виконують завдання, для яких вони були спроектовані, але також визначають нові стандарти в технологічному просторі, вносячи величезний внесок у визначення майбутнього розвитку технологій.

ТЕМА 1. ОСОБЛИВОСТІ ВБУДОВАНИХ СИСТЕМ

План лекції

- Класифікація вбудованих систем
- Прикладні області вбудованих систем
- Основні характеристиви
- Загальна структура вбудованих систем
- Основні відмінності мікроконтролерів від мікропроцесорів
- Архітекрура обчислювальних систем
- Архітектура системи команд
- Типи мікроконтролерів та вибір мікроконтроллерів
- Кар'єрні можливості Embedded-розробника

Текстова частина лекції

Класифікація вбудованих систем

Вбудовані системи можна класифікувати за декількома ключовими критеріями, враховуючи різні аспекти їх призначення, структури та функціональності:

1. За призначенням/застосуванням: автомобільні, медичні, побутові тощо.

2. За апаратно-програмним забезпеченням:

- Мікроконтролерні вбудовані системи – використовують мікроконтролери, зазвичай для простих застосунків з обмеженими ресурсами.
- Мікропроцесорні вбудовані системи – засновані на мікропроцесорах, забезпечують більші обчислювальні можливості та гнучкість у програмуванні.

3. За співвідношенням інформаційних і керуючих функцій:

- Інформаційні системи – системи збору та обробки даних.
- Керуючі системи – системи автоматичного керування.

4. За типом системи:

- Реального часу (*RTOS*) – системи, які повинні реагувати на події в обмежені строки. Вони використовуються у сферах, де важлива точність інтервалів часу, наприклад, у сфері авіації або автомобільної безпеки.
- Загального призначення – системи, для яких точність таймінгу не критична. Наприклад, побутові електроприлади.

5. За співвідношенням обчислювальної і комунікаційної складової:

- Системи з важливою обчислювальною складовою – обробка даних.
- Системи з акцентом на комунікаційні функції – ввід-вивід даних.

6. За типом енергозабезпечення:

- З живленням від мережі.
- Автономні системи, що живляться власними джерелами енергії (батареї, сонячні елементи).

7. За способом комунікації:

- Проводові вбудовані системи – взаємодіють за допомогою дротових з'єднань.
- Бездротові вбудовані системи – взаємодіють за допомогою бездротових технологій, таких як *Wi-Fi*, *Bluetooth*, *NFC*.

8. За розміром та складністю:

- Малий розмір та прості системи – розумні сенсори для фіксування одного чи декількох показників.
- Великий розмір та складні системи – системи в автомобілях, літаках, промислових установках.

9. За ступенем участі людини:

- Автоматичні системи – без участі людини в операційному процесі.
- Автоматизовані системи – часткова або повна участь людини у процесі.

Ці критерії не є взаємовиключними, і конкретна вбудована система може відповідати декільком класифікаційним критеріям одночасно.

Прикладні області вбудованих систем

Вбудовані системи присутні в різних аспектах життя. Далі наведено декілька прикладів систем, що використовуються у різних галузях і мають різноманітні функції:

- **Автомобільні системи управління** – електронні системи управління двигуном, системи стабілізації, антиблокувальна система (ABS).
- **Медичні пристрої** – електрокардіографи, імплантовані пристрої для моніторингу здоров'я.
- **Системи вбудованого контролю в літаках** – автопілоти, системи контролю безпеки.
- **Побутова техніка** – смарт-телевізори, холодильники з інтелектуальним управлінням.
- **Мережеві пристрої** – маршрутизатори, комутатори, пристрої для мережевої безпеки.
- **Електроніка для розваг** – вбудовані системи в гральних консолях, аудіосистеми.
- **Контроль та управління в промисловості** – програмовані контролери для автоматизації виробничих процесів, системи моніторингу та контролю великих механізмів.
- **Безпека та відеоспостереження** – системи відеоспостереження, контролю доступу.
- **Інтернет речей (IoT)** – системи домашньої автоматизації, розумні термостати, вбудовані сенсори у пристроях.

Основні характеристиви

Вбудовані системи відрізняються від загальних обчислювальних систем своєю специфікою та особливостями. Основні характеристики вбудованих систем включають:

- **Надійність**. Вбудовані системи, як правило, повинні працювати стійко і безперебійно, оскільки їх використовують у критичних застосунках (медичні пристрої, автомобільна безпека, системи керування літаком).
- **Вмістимість та обсяг пам'яті.** Часто присутнє обмеження в обсязі пам'яті, який може варіюватися в залежності від потреб для застосування системи.

- **Енергоефективність**. Багато вбудованих систем працюють в областях, де обмежені джерела живлення, тому енергоефективність є дуже важливою для енергозберігання та для забезпечення тривалої роботи від батарей.
- **Робота з реальним часом.** Деякі вбудовані системи потребують обробки даних та реагування на події в дуже обмежені терміни. Використання операційних систем реального часу дозволяє гарантувати точність та своєчасність обробки інформації.
- **Керованість** – можливість взаємодії та керування вбудованими системами через зовнішні інтерфейси або мережу.
- **Можливість розширення**. Ця характеристика вказує на здатність вбудованої системи до оновлення або додавання нового функціоналу.
- **Вартість** – у багатьох випадках, вбудовані системи повинні бути не тільки ефективними, а і економічно доцільними.

Ці характеристики допомагають визначити, чи вбудовані системи відповідають вимогам конкретних застосувань та середовища їх експлуатації.

Загальна структура вбудованих систем

Структура вбудованих систем може варіюватися в залежності від конкретного застосування та вимог до системи. Однак існують деякі загальні компоненти та елементи, які часто зустрічаються в більшості вбудованих систем. Нижче наведено основні компоненти та їх функції:

1. **Центральний процесор (CPU)**. Відповідає за виконання інструкцій програм, обчислень, управління роботою інших компонентів та роботу усієї систем вцілому. Центральний процесор (CPU) вбудований у мікроконтролер.
2. **Пам'ять**. Відповідає за зберігання програмного забезпечення, даних та проміжних результатів обчислень. Приклад: флеш-пам'ять для програм та оперативна пам'ять (RAM) для тимчасового зберігання даних під час виконання програми.
3. **Інтерфейси, комунікаційні/мережеві модулі, контролери введення/виведення (I/O controllers).** Відповідають за забезпечення комунікації між компонентами однієї вбудованої системи або між вбудованою системою та зовнішнім середовищем, іншими пристроями чи мережами. Приклад: порти введення/виведення (I/O ports), USB, UART, SPI, I2C, GSM/3G/4G/5G, Wi-Fi, Bluetooth, Zigbee, LoRa, NB-IoT.
4. **Датчики та периферійні пристрої.** Відповідають за збір інформації про стан системи та оточуючого середовища, а також про дії користувача, за наявності. Приклад: акселерометри, гіроскопи, температурні сенсори.
5. **Електроживлення.** Відповідає за забезпечення електропостачання до всіх компонентів вбудованої системи. Приклад: блоки живлення, батареї, акумулятори, сонячні панелі та інші джерела енергії.
6. **Система керування живленням.** Відповідає за оптимізацію споживання енергії для підтримки довготривалої роботи від батареї або в обмежених енергетичних умовах. Приклад: регулятори напруги, системи управління енергією та інші елементи.
7. **ПЗ та/або операційна система.** Відповідає за керування ресурсами, обробку завдань та забезпечення взаємодії із користувачем чи іншими системами. Приклад: програми

написані на різних мовах програмування, спеціалізовані бібліотеки, операційні системи загального призначення та реального часу, спеціалізоване програмне забезпечення.

8. **Механічні компоненти.** Відповідають за забезпечення фізичної стійкості, забезпечення руху та позиціонування, тепловідведення та охолодження, захист від зовнішніх впливів та інше. Приклад: корпуси та панелі, кріплення та кріпильні елементи, актуатори та інші механічні складові, які можуть бути необхідними в залежності від призначення вбудованої системи.

Всі ці компоненти працюють разом для забезпечення функціональності та продуктивності вбудованої системи. Розробники вбудованих систем враховують специфічні вимоги та обмеження, щоб оптимізувати дизайн та досягти бажаних характеристик.

Відмінності мікроконтролерів від мікропроцесорів

Інтегрована схема (ІС) — це електронний компонент, який складається з великої кількості інших електронних компонентів (таких як транзистори, резистори, конденсатори), що розташовані на одному кремнієвому кристалі (або іншому напівпровідниковому матеріалі).

Основні характеристики інтегральних схем:

- **Кількість компонентів**. ІС може містити від декількох десятків до мільярдів елементів, залежно від типу та призначення.
- **Технологія виготовлення** визначає розмір компонентів та їхню густину.
- **Функціональність**. ІС може виконувати різноманітні функції, від обчислення і управління до обробки сигналів та зберігання інформації.
- **Потужність**. Вимірюється ватами і визначає, скільки електричної енергії споживає ІС під час роботи.
- **Напруга живлення**, яка подається на ІС для її правильної роботи.
- **Тип корпусу** – фізична оболонка, яка оточує кристал і забезпечує його захист та з'єднання з іншими елементами системи.

Інтегральні схеми дозволяють створювати складні електронні пристрої у вигляді мініатюрних чіпів, що значно полегшує їхнє використання та інтеграцію в інші пристрої.

Мікроконтролери та мікропроцесори є двома різними типами інтегральних схем, які мають різні характеристики та використовуються для різних цілей.

Мікропроцесор – це інтегрована схема, яка містить основні обчислювальні та управляючі блоки, такі як арифметико-логічний пристрій (АЛП), управляючий пристрій (УП) та регістри. Основні характеристики мікропроцесора включають архітектуру, тактову частоту, кількість ядер, розмір кеш-пам'яті, набір інструкцій та інші параметри. Вони виконують різні програми, забезпечуючи функціональність пристроїв, в які будуть вбудовані.

Особливості мікропроцесора:

- **Застосування**. Використовується для виконання різноманітних завдань (обчисленнь) загального призначення (в персональних комп'ютерах, серверах, ноутбуках та інших системах).
- **Периферійні пристрої.** Зазвичай має пристрої, що відповідають за введення та виведення інформації.
- **Пам'ять**. Вимагає зовнішньої пам'яті для зберігання програмного коду та даних.

- **Розмір та потужність.** Зазвичай має більший за мікроконтролер розмір та може споживати більше енергії, оскільки призначений для виконання великої кількості різноманітних завдань.

Мікроконтролер – це інтегрована схема, яка включає в себе не лише обчислювальний блок (мікропроцесор), а також вбудовану пам'ять (програмну та даних), периферійні пристрої для взаємодії з навколишнім середовищем, аналогові та цифрові порти введення/виведення, таймери, засоби комунікації та інші блоки.

Особливості мікроконтролера:

- **Застосування**. Використовується для керування вбудованими пристроями і системами (розумні пристрої Інтернету речей, автомобільні системи, промислові контроллери та інше).
- **Периферійні пристрої**. Для спрощення взаємодії із навколишнім середовищем може мати значну кількість вбудованих периферійних пристроїв, таких як: порти введення/виведення, АЦП, UART, Wi-Fi та інше.
- **Пам'ять**. Мають вбудовану флеш-пам'ять для зберігання програмного коду та EEPROM або інші види пам'яті для зберігання даних.
- **Розмір та потужність**. Невеликий розмір та споживають небагато енергії, оскільки призначені для використання в енергоефективних вбудованих системах.

Обираючи між мікроконтролерами та мікропроцесорами, важливо враховувати специфічні вимоги конкретної задачі та характеристики системи, яку потрібно розробити. Мікроконтролери часто використовуються у вбудованих системах, де потрібна детермінованість та енергоефективність, тоді як мікропроцесори забезпечують більшу універсальність для широкого спектру завдань.

Архітекрура обчислювальної системи

У контексті мікроконтролерів та мікропроцесорів, термін "архітектура" вказує на загальний дизайн та організацію інтегральних схем. Архітектура визначає, як вони виконують операції обчислення, як організована пам'ять, як взаємодіють з периферійними пристроями та як керують внутрішніми та зовнішніми змінними.

Дві ключових архітектурам в світі обчислювальних систем – Гарвардська та Фон Нейманівська. Кожна з цих архітектур визначає структуру та організацію елементів системи. Розуміння їх особливостей має важливе значення для інженерів та розроб ників.

Фон Нейманівська архітектура

Основна ідея – об'єднання пам'яті для програм та даних. Одна спільна шина для читання/запису. Зустрічається у мікропроцесорах та комп'ютерах, де важлива універсальність та програмна гнучкість.

***Переваги*:**

- **Ефективність**. Ефективніше використання апаратних ресурсів.
- **Компактність**. Зазвичай потрібно менше елементів для реалізації.

Недоліки: У складних системах може виникнути обмеження швидкодії через об'єднання шин для програм та даних.

Гарвардська архітектура

Основна ідея – відокремлення пам'яті для програм та даних. Тобто, є окремі шини для інструкцій програми та даних. Часто використовується в мікроконтролерах та вбудованих системах, де важлива швидкодія та простота системи.

***Переваги*:**

- **Швидкість**. Одночасне зчитування/запис програм і даних може покращити швидкодію.
- **Простота**. Легше впроваджувати та програмувати.

Недоліки: Потрібно більше апаратних засобів для реалізації окремих шин.

Архітектура системи команд

Система команд (ISA - Instruction Set Architecture) — це набір операцій та форматів команд, які підтримується конкретною архітектурою процесора або мікроконтролера. Вона визначає, які команди може виконувати процесор, як вони представлені у вигляді бітових кодів (кожна має свій унікальний код), та як впливають на його роботу. Наприклад, система команд може включати низку функцій, таких як: арифметичні та логічні операції; операції з пам'яттю; управління переходами та інші.

Типи систем команд в сучасних мікроконтролерах:

- **CISC (Complex Instruction Set Computing)**. Включає велику кількість складних та різноманітних команд, які можуть виконувати багато різних операцій. Приклади: архітектура x86 (Intel 80x86), ARM в режимі ARM.
- **RISC (Reduced Instruction Set Computing).** Має скорочений набір команд, що спрощує їх виконання та підвищує швидкість і енергоефективність. Приклади: архітектура ARM в режимі Thumb, MIPS, RISC-V.
- **VLIW (Very Long Instruction Word).** В довгу інструкцію включається одразу декілька операцій, які виконуються паралельно. Вимагає від компілятора більш активної ролі у генерації коду. Приклади: Intel Itanium.
- **EPIC (Explicitly Parallel Instruction Computing)**. Схожа на VLIW, але з більшою кількістю паралельних одиниць виконання команд. Приклади: Intel Itanium (IA-64).
- **DSP (Digital Signal Processor).** Спеціалізовані системи команд для обробки цифрових сигналів, зокрема для високошвидкісної математичної обробки. Приклади: сімейство TMS320 (Texas Instruments), ADSP (Analog Devices).
- **SIMD (Single Instruction, Multiple Data)**. Одна інструкція виконує ту саму операцію над багатьма елементами даних одночасно. Приклади: архітектура SSE/AVX (Intel), NEON (ARM).

Слід зазначити, що "RISC", "CISC" і т.д., можуть вказувати на різні речі залежно від контексту. Ці терміни використовуються як для опису системи команд, так і для архітектури процесорів, які таку систему команд використовують.

Типи мікроконтролерів

Мікроконтролери можна класифікувати за розрядністю, яка визначається кількістю бітів у центральному процесорі, основні з яких це:

- **8-розрядні**. Призначені для реалізації систем середньої або малої продуктивності та мають просту систему команд. Зазвичай використовуються в пристроях з невеликою складністю. Представники цього типу включають: STM8, ATmega328p, PIC16, Motorola HC05 – HC012, MCS – 51 (Intel) та інші.
- **16-розрядні**. Адаптовані для швидшої реакції на зовнішні події та можуть справлятись з більш складними завданнями. Забезпечують точність та швидкість управління, що є важливим в таких застосуваннях, як промислові роботи, автомобільна промисловість тощо. Приклади: ST10, PIC24, C161-C167 (Siemens), HC16 (Motorola), MCS96/196/296 (Intel).
- **32-розрядні**. Такі МК здатні обробляти більші обсяги даних та виконувати більш складні операції. Вони часто використовуються в системах, де потрібна високошвидкісна обробка даних, наприклад для: обробки сигналів у системах телефонії, аудіо- та відеообробки, систем передачі інформації та іншого. Приклади: STM32, ESP32, RP2040 (Raspberry Pi Pico) та інші.

Також, широкий спектр мікроконтролерів можна класифікувати за сімействами, відповідно до виробників або архітектур**:**

- **AVR** (Atmel). Модифікована Гарвардська архітектура, RISC. Прості у використанні, ефективні для великої кількості вбудованих застосувань, включаючи прості DIY-проекти та освітній сектор (Arduino).
- **PIC** (Microchip). Модифікована Гарвардська архітектура, RISC. Великий вибір моделей з різними функціональними можливостями, широке використання у вбудованих системах, медичній техніці, автомобільній електроніці.
- **STM32** (STMicroelectronics). Побудовані на ядрах ARM Cortex-M. Є варіації як з Фон Нейманівською так і з Гарвардською архітектурою, RISC. Великий вибір моделей, різні можливості введення/виведення, розширена підтримка периферійних пристроїв, використовується в промисловій автоматизації, електроніці для автомобілів, пристроях Інтернету речей (IoT) та інших застосуваннях.
- **ESP8266/ESP32** (Espressif). Гарвардська архітектура, система команд Xtensa (в ESP8266), RISC-V (в ESP32). Через вбудований Wi-Fi, популярні у спільноті розробників проектів IoT.
- **RP2040** (Raspberry Pi Ltd.). Ядро ARM Cortex-M0+. Доступні та популярні мікроконтролери від Raspberry Pi. Застосовуються в різноманітних варіантах плат розробника від цієї компанії.
- **MSP430** (Texas Instruments). Фон Нейманівська архітектура, RISC. Низьке споживання енергії, використовується в пристроях з живленням від батарей, таких як датчики, пристрої IoT.
- **8051** (Intel, Atmel, і інші). Модифікована Гарвардська архітектура, CISC. Оригінально розроблене Intel у 1980 році, але досі можна зустріти клони від інших виробників. Мікроконтролера даного сімейства широко використовувався в промисловості,

автомобільній електроніці, системах управління, світлофорах, датчиках та багатьох інших пристроях.

- **HC11/HC12** (Motorola/Freescale/NXP). Фон Нейманівська архітектура, CISC. Сімейство представлене в середині 90-х, використовувалось в багатьох вбудованих системах, включаючи автомобільну електроніку, медичні прилади та інші пристрої.
- **68k** (Motorola/Freescale). Фон Нейманівська архітектура, CISC. Сімейство було дуже популярним у 80-90х роках та використовувалось у різноманітних областях.
- **Z80** (Zilog). Фон Нейманівська архітектура, CISC. Можна знайти в ретро комп'ютерах, вбудованих системах та військових пристроях. Сімейство домінувало на ринку з середини 70-х до середини 80-х років.

Ці сімейства представляють лише малу частину ринку мікроконтролерів. Кожне має свої властивості та застосування, а вибір конкретного мікроконтролера, що буде оптимально відповідати вимогам та завданням, зазвичай являється доволі складною процедурою.

Вибір мікроконтроллерів

Вибір мікроконтролера – це ключовий етап у розробці будь-якого електронного пристрою або системи. Підбір мікроконтролера повинен враховувати конкретні потреби та обмеження проекту, забезпечуючи оптимальну продуктивність та ефективність для конкретних завдань. При виборі МК слід звернути увагу на наступні характеристики та особливості:

1. **Властивості процесора:**
   - **Архітектура**. RISC або CISC і т.д., залежно від потреб проекту.
   - **Швидкість.** Тактова частота процесора.
   - **Кількість ядер.** Важливо для паралельної обробки завдань.
   - **Розмір слова**. Визначає кількість бітів, які обробляються одночасно. Більший розмір слова може покращити швидкість обчислень та точність. Зазвичай це 8, 16, 32 або 64 біти.
   - **Підтримка плаваючої коми**. Якщо додатки вимагають великої точності арифметичних операцій або обробки дробових чисел, обирайте мікроконтролери з підтримкою апаратних операцій з плаваючою комою.
   - **Підтримка передбачення гілок (branch prediction).** Дозволяє заздалегіть підготувати потрібні дані для оптимізації виконання програми.
2. **Пам'ять:**
   - **Підтримка кешу інструкцій та даних**. Кеш – це швидкодіюча пам’ять, яка зберігає нещодавно використані дані та інструкції. Підтримка кешу допомагає прискорити виконання програм.
   - **Обсяг Flash та RAM.** Flash-пам’ять використовується для зберігання програмного коду, а RAM – для зберігання даних під час виконання програми. Більший обсяг Flash і RAM дозволяє розробникам створювати більш потужні програми.
   - **EEPROM**. Використовується для зберігання даних після вимкнення живлення. Пам'ять EEPROM може бути використана для довготривалого зберігання налаштувань, лічильників, ідентифікаторів пристроїв та інших даних.

- **Кількість регістрів** (швидка пам'ять, але їх мало). Регістри – це невеликі швидкодіючі блоки пам’яті, які використовуються для зберігання проміжних результатів обчислень. Кількість регістрів в мікроконтролері впливає на його можливості та продуктивність.
- **Підтримка DMA (Direct Memory Access)**. DMA дозволяє пристроям взаємодіяти безпосередньо з пам’яттю, без участі процесора, що полегшує обробку великих обсягів даних та звільняє процесор для інших завдань покращуючи продуктивність системи.

3. **Периферійні пристрої:**
   - **I/O порти.** Кількість та можливості портів для підключення до зовнішніх пристроїв.
   - **Комунікаційні інтерфейси.** UART, SPI, I2C, USB для взаємодії між мікросхемами та з іншими пристроями.
4. **Живлення:**
   - **Діапазон напруги.** Визначає, які напруги живлення може приймати мікроконтролер.
   - **Споживана потужність.** Важливо для портативних і акумуляторних пристроїв.
5. **Доступність та вартість:**
   - **Вартість мікроконтролера.** Впливає на бюджет розроблюваного проекту.
   - **Доступність на ринку.** Впливає на можливості отримання мікроконтролерів для виробництва в потрібних кількостях.
6. **Енергоефективність та режими енергозбереження**. Для заощадження енергії, особливо важливо для автономних пристроїв.
7. **Розробницькі інструменти та підтримка:**
   - **Наявність SDK (Software Development Kit).** Впливає на легкість розробки програмного забезпечення.
   - **Підтримка спільноти та документація**. Допомагає вирішувати проблеми та отримувати додаткову інформацію.
8. **Вимоги до промислової безпеки та надійності:**
   - **Температурний діапазон.** Якщо пристрій використовується в екстремальних умовах.
   - **Наявність вбудованих перевірок та механізмів для уникнення помилок.**
9. **Розмір та форм-фактор.** Фізичний розмір мікроконтролера особливо важливий для пристроїв з обмеженим внутрішнім простором.
10. **Інші спеціальні властивості.** Наприклад, наявність вбудованих периферійних пристроїв, АЦП, ШІМ-таймерів, датчиків.

Документація відіграє важливу роль під час вибору мікроконтролера. Виробники чіпів зазвичай надають такі типи документів:

- **Селектор продуктів виробника (Selector guide)** допомагає обрати правильну мікроконтролерну або мікропроцесорну платформу для конкретного застосування. Він зазвичай містить порівняння різних моделей по параметрах, таких як швидкість, обсяг пам'яті, кількість виводів та інші характеристики.

- **Короткий опис продукту (Product Brief)** надає загальний огляд мікроконтролера, включаючи його основні характеристики, функції та переваги. Він допомагає швидко зрозуміти, чи відповідає дана платформа потребам.
- **Технічний опис (Datasheet)** – це найважливіший документ для оцінки мікроконтролера. В ньому міститься докладна інформація про всі параметри, характеристики та функції пристрою, включаючи електричні та часові параметри, блок-схеми, реєстри, розподіл пам'яті, фізичні розміри та інші важливі дані.
- **Опис контактів (Pinout)** містить інформацію про розташування та функціональне призначення кожного виводу мікроконтролера. Його використовують для планування підключення зовнішніх пристроїв та плат.
- **Посібник з програмування мікроконтролера (Programming manual)** описує інструкції та приклади коду для розробки програмного забезпечення для конкретної мікроконтролерної платформи. Він допомагає розібратися у специфіці програмування пристрою та ефективно використовувати його можливості.
- **Помилки мікросхеми (device errata)** – документ, що містить інформацію про відомі помилки або проблеми в роботі чіпа, а також коригувальну інформацію та механізм вирішення знайдених проблем.

Плати розробника

Мікроконтролер – це конкретний чіп, для практичного застосування якого зазвичай потрібні додаткові компоненти.

Плата розробника – це спеціальна плата, яка призначена для швидкого прототипування та розробки вбудованих систем. На ній, крім мікроконтролера чи мікропроцесора, зазвичай розміщені такі компоненти, як: регулятори та стабілізатори напруги, кварцовий резонатор або генератор, роз'єми для підключення до комп'ютера, переферійні пристрої, індикатори та кнопки, засоби для програмування (програматори) та відлагодження. Така плата дозволяє розробникам швидко перевіряти свої ідеї та доводити концепції.

Доказ концепції – це процес випробування нової ідеї або концепції для визначення її життєздатності. Він включає створення прототипу, його випробування на практиці та оцінку результатів. Це дуже важливий етап перед початком масштабної розробки, оскільки він дозволяє виявити та вирішити можливі проблеми та недоліки ще на ранніх етапах проекту.

Найпопулярнішими платами розробника для вбудованих систем є ті, що пропонують гнучкість, широкий спектр використання та підтримку різних операційних систем. Ось деякі з них:

- **Arduino** – широко використовується для прототипування та освітніх цілей, підтримує різноманітні модулі та сенсори. Прості у використанні та мають широкий вибір розширювальних модулів (шілдів).
- **Raspberry Pi** – потужна одноплатна комп'ютерна система, яка може використовуватися для великої кількості застосувань, включаючи проекти Інтернету речей, робототехніку, медіацентри та інше.
- **ESP32** – плати з мікроконтролерами від Espressif. Вони мають вбудований Wi-Fi та Bluetooth та ідеально підходять для проектів Інтернету речей.

- **STM32 Discovery** – це серія плат, що використовує мікроконтролери STM32 і надає розширені можливості для розробки вбудованих систем з великою кількістю периферійних пристроїв та інтерфейсів.
- **BeagleBone** – сімейство одноплатних комп'ютерів з ARM-процесорами, які працюють на базі Linux.
- **Jetson –** серія плат від NVIDIA, які мають високу обчислювальну потужність і призначені для застосувань штучного інтелекту, машинного навчання, аналізу даних, обробки зображень, розпізнавання образів та іншого.
- **NXP Semiconductors** пропонує широкий спектр плат розробника, які підтримують різноманітні застосування від автомобільної промисловості до IoT та індустріальних рішень.

Загалом, вибір плати розробника залежить від специфічних вимог проекту та концепції, яку необхідно довести.

Кар’єрні можливості в сфері вбудованих систем

У сфері вбудованих систем існує багато цікавих напрямків для спеціалістів з різними фаховими напрямками. Ось деякі з них:

- **Розробники програмного забезпечення (Embedded Software Developer)** спеціалізуються на написанні прикладних програм, драйверів, мережевих стеків, бібліотек та інших компонентів, які взаємодіють з користувачем або іншим програмним забезпеченням.
- **Розробники мікропрограмного забезпечення (Embedded Firmware Developer)** працюють на нижчому рівні абстракції, ближче до апаратного забезпечення, включаючи початкове завантаження системи, налаштування апаратних ресурсів, керування периферійними пристроями та ішше.
- **Інженери з забезпечення якості та тестування (Embedded Systems QA/Test Engineer)** відповідальні за створення та проведення тестів для виявлення дефектів та забезпечення якості системи.
- **Інженери вбудованих систем (Embedded Systems Engineer)** мають ширший спектр обов'язків, які включають як вибір/проектування апаратного забезпечення, так і розробку програмного, а також інтеграцію та налагодження системи вцілому.
- **Інженери в області досліджень та розробок (Embedded Research and Development Engineer, R&D)** займаються вивченням нових технологій та розробкою інноваційних рішень.
- **Спеціалісти з безпеки вбудованих систем (Embedded Security Engineer)** працюють над захистом від атак та зловживань, враховуючи різноманітні вразливості та заходи безпеки.

Такі спеціалісти можуть працювати в різних секторах, включаючи: телекомунікації, транспорт, промислове устаткування, медична техніка, побутова апаратура, енергетика та багато інших.

Запитання для самоперевірки

1. Назвіть основні тенденції розвитку вбудованих систем у сучасному світі.
2. Які прикладні області використання вбудованих систем ви знаєте?
3. Які основні характеристики вбудованих системи?
4. Що входить в загальну структуру вбудованих систем?
5. В чому полягають основні відмінності між мікроконтролерами та мікропроцесорами?
6. Що таке архітектура вбудованих систем і чому вона важлива?
7. Які системи команд існують в мікроконтролерах?
8. Назвіть різні типи мікроконтролерів та їх призначення.
9. Які характеристики важливі при виборі мікроконтролера для конкретного проекту?
10. Що таке плати розробника і яку роль вони відіграють у процесі розробки вбудованих систем?
11. Які кар'єрні можливості у сучасному світі існують в сфері вбудованих систем?

Тести

1. Що таке вбудовані системи?
   A) Системи, що спроектовані для виконання конкретних завдань або функцій.
   B) Системи, що використовуються на будівництві.
   C) Системи, які забезпечують автономну роботу підключених пристроїв.
   D) Системи, що вбудовуються в мобільні пристрої.
2. Сучасні тенденції вбудованих систем:
   A) Збільшення розміру вбудованих систем.
   B) Збільшення рівня автоматизації в різних сферах.
   C) Заміна вбудованих систем на загальні мікропроцесори.
3. Які бувають вбудовані системи за способом комунікації?
   A) Автомобільні.
   B) Мікроконтролерні.
   C) З живленням від мережі.
   D) Бездротові.
4. Прикладні області вбудованих систем включають:
   A) Медичні пристрої.
   B) Автомобільну промисловість.
   C) Телекомунікації.
   D) Усі зазначені варіанти вірні.
5. Які основні характеристики вбудованих систем?
   A) Розмір, швидкодія та безпека.
   B) Вага, ціна та енергоспоживання.
   C) Надійність, енергоефективність та керованість.
   D) Виробник, кількість кнопок, потужність.

6. Що таке мікропроцесор?
   A) Пристрій для виконання обчислювальних операцій.
   B) Вбудований мікрочіп для зберігання інформації.
   C) Програмована мікросхема для роботи з графікою.
   D) Електронний пристрій для роботи з мікро процесами.
7. Чим відрізняється мікроконтролер від мікропроцесора?
   A) Мікроконтролер має вбудовану пам'ять та інші модулі, що дозволяють йому працювати незалежно, у той час як мікропроцесор вимагає зовнішніх компонентів для роботи.
   B) Мікропроцесор призначений для виконання операцій обробки великих об'ємів даних, тоді як мікроконтролер – для обробки малого об'єму даних.
   C) Мікропроцесор може працювати тільки у вбудованих системах, тоді як мікроконтролер може використовуватися в різних областях.
   D) Мікроконтролер і мікропроцесор – це синоніми, тому вони нічим не відрізняються.
8. Оберіть основні архітектури обчислювальної техніки:
   A) Логічна та фізична.
   B) Гарвардська та Фон Нейманівська.
   C) AMD та Intel.
   D) Гарвардська та Оксфордська.
   E) Фон Бісмаркська та Кембриджська.
9. Які особливості Гарвардської архітектури?
   A) Програми та дані зберігаються в одній області пам'яті.
   B) Використовуються різні види пам'яті для зберігання програм та даних.
   C) Має 3 шини даних.
   D) Дозволяє підібрати оптимальний обсяг пам'яті.
10. За що відповідає розрядність мікроконтролера?
    A) Кількість підтримуваних інтерфейсів.
    B) Кількість можливих операцій обчислення за один такт.
    C) Кількість бітів, яку може обробляти мікроконтролер за один цикл.
    D) Швидкість передачі даних через Інтернет.
11. Які сімейства мікроконтролерів існують?
    A) Intel, AMD, NVIDIA.
    B) AVR, PIC, STM32.
    C) PlayStation, Xbox, Nintendo.
    D) Apple, Samsung, Huawei.
12. Які найважливіші особливості мікроконтролера?
    A) Виконання корпусу та розмір пам'яті.
    B) Кількість портів та швидкість пам'яті.
    C) Кількість регістрів, розмір слова та розмір пам'яті.
    D) Розмір слова, розмір речення, кількість виводів.

13. Чим відрізняється Datasheet від посібника для програмування?
    A) Datasheet містить характеристики пристрою, тоді як посібник для програмування містить інструкції щодо написання коду.
    B) Datasheet містить загальну інформацію, тоді як посібник для програмування надає детальніший опис цієї інформації та алгоритмів.
    C) Datasheet надає дані про фізичні габарити пристрою, а посібник – про його електричні характеристики.
14. Для чого потрібні плати розробника та різноманітні модулі?
    A) Для доведення математичних теорем на практиці.
    B) Для спрощення виробництва вбудованих систем.
    C) Для розробки прототипів та доказу концепції.
    D) Для підключення до мережі Інтернет.

ТЕМА 2. ОСОБЛИВОСТІ ПРОГРАМУВАННЯ ВБУДОВУВАНИХ СИСТЕМ

План лекції

- Ключові етапи та аспекти розробки ПЗ для вбудованих систем
- Програмне навантаження
- Різновиди програмування вбудованих систем
- Операційні системи
- Види коду
- Мови програмування
- Цикл виконання інструкцій (Fetch-Decode-Execute)
- Синтаксис C/C++
- Процес збірки програми

Текстова частина лекції

Програмування вбудованих систем – це процес створення програм, які виконуються на мікроконтролерах чи мікропроцесорах. Цей процес відрізняється від розробки ПЗ для звичайних комп'ютерів, оскільки вбудовані системи зазвичай мають обмежені ресурси, а їхнє програмне забезпечення орієнтоване на виконання спеціалізованих завдань.

Ключові етапи циклу розробки ПЗ для вбудованих систем включають:

- **Визначення вимог.** Розробка починається з аналізу вимог до системи, що включає визначення функціональних та нефункціональних вимог, таких як продуктивність, швидкодія, обмеження ресурсів, надійність тощо.
- **Вибір апаратної платформи.** Є ключовим етапом, де визначається на якому мікроконтролері чи процесорі буде працювати програмне забезпечення.
- **Розробка коду (програмування).** Це етап, де розробники використовують мови програмування, такі як C, C++, асемблер та інші, для створення програм, які виконують специфічні завдання.
- **Відлагодження та тестування.** Важливий етап, де знаходяться та вирішуються різноманітні програми. Для відлагодження коду та виконання тестування зазвичай використовуються спеціальні інструменти.
- **Оптимізація.** За потреби, виконується для зменшення використання ресурсів і покращення продуктивності.
- **Завершення та випуск.** Після успішного відлагодження і тестування програмне забезпечення готове до випуску. Його встановлюють на вбудовані пристрої для використання в реальних умовах.
- **Підтримка та оновлення.** В процесі експлуатації системи надаються консультації клієнтам, виправляються можливі помилки або вводяться певні покращення чи додавання нових функцій.
- **Документування**. Під час розробки важливо створювати документацію для коду та системи. Це сприяє розумінню та підтримці системи в майбутньому.

Розробка ПЗ для вбудованих систем, крім навичок програмування, вимагає також глибоких знань апаратного забезпечення, розуміння обмежень вбудованих пристроїв та оптимізації їх продуктивності.

Різновиди програмування вбудованих систем

Розробка коду (програмування) – це ключовий етап циклу розробки, оскільки саме тут створюється програма, яка визначає поведінку вбудованої системи. Можна виділити три основних різновиди програмування для вбудованих систем:

- **Bare-metal програмування** («на голому металі») – це метод програмування, при якому розробник створює програмне забезпечення без використання операційної системи або інших високорівневих середовищ. Таке програмування часто використовують для прямої взаємодії з апаратним забезпеченням, отримання максимальної продуктивності та ефективності ресурсів.
- **Програмування під RTOS (Real-time Operating System) –** означає розробку програмного забезпечення з використанням операційної системи реального часу (RTOS). RTOS дозволяє розробникам створювати програми, які мають гарантований час виконання та можуть ефективно обробляти завдання в режимі реального часу. У такому програмуванні важливо враховувати особливості системи реального часу, такі як точність виконання завдань, механізми планування, обробка подій у визначений часовий інтервал та інші. Прикладами популярних RTOS є FreeRTOS, TinyOS, Contiki.
- **Програмування з використанням Embedded GPOS (General Purpose Operating System)** відноситься до розробки ПЗ з використанням операційної системи загального призначення. Можливості GPOS щодо гарантій часу відповіді обмежені, тому вони не дуже пристосовані до роботи в режимі реального часу, хоча такі системи можуть забезпечити більше функціональності. Прикладами «вбудованих» GPOS є Linux, Windows Embedded, Android (в певних випадках).

Надлишкові програмні витрати

Software overhead (надлишкові програмні витрати) – це додаткове програмне навантаження, що потребується для виконання програмного забезпечення, крім виконання основних завдань. Його наявність може призвести до збільшення часу виконання програми, збільшення використання пам'яті або енергії вцілому.

Надлишкові програмні витрати можуть виникати через різні фактори, наприклад, використання операційних систем з великим функціоналом, високорівневих мов програмування, абстракцій та інших програмних шарів, які додаються для спрощення розробки, але при цьому можуть призвести до втрати продуктивності.

Зазвичай кількість software overhead намагаються звести до мінімума, особливо у вбудованих системах та додатках, де кожна операція може бути критично важливою.

Особливості кожного з видів програмування

Програмування «на голому металі»:

- **Мало чи жодного software overhead.** Що дозволяє досягти високої продуктивності та ефективно використовувати ресурси.

- **Прямий контроль над апаратурою.** Дозволяє розробнику оптимізувати програмне забезпечення, виконувати завдання ефективніше і, наприклад, використовувати для цього менш потужні мікроконтролери.
- **Низька витрата енергії.** Без операційної системи, яка використовує ресурси, вбудована система може працювати енергоефективніше, що є критичним для багатьох застосувань.
- Зазвичай використовується для розробки програм, що служать одній конкретній меті або працюють з конкретною апаратурою.

Програмування під RTOS:

- **Багатопотоковість**. Різні задачі, такі як: робота з давачами, відображення інформації, обмін даними через мережу, можуть виконуватися паралельно.
- **Детермінізм**. Система здатна виконувати завдання з передбачуваним і повторюваним часом відгуку.
- **Витрати ресурсів на планування**. Ресурси витрачаються на постійну фонову роботу операційної системи, яка займається ефективним розподілом часу для виконання різних завдань. Може вимагати більш потужного мікроконтролера.
- **Простіше переносити програму на інші платформи**. Використання загальних бібліотек та поділ програми на незалежні модулі (завдання), дозволяє доволі легко переносити її на будь-який інший мікроконтролер, що працює з тією ж RTOS.
- **Є можливість керувати обладнанням напряму**, шляхом маніпулювання регістрами, але наявність таких рішень зазвичай робить код менш переносним.

Програмування під Embedded GPOS:

- **Керування багатьма процесами і потоками одночасно.** Застосовується в вбудованих системах із багатьма складними завданнями, такими як виконання мережевих операцій, робота з файловими системами, надання графічних інтерфейсів користувачам тощо.
- **Не гарантується точний час виконання завдань.** Система може бути не здатна забезпечити негайну реакцію на події з високою точністю.
- **Великі програмні витрати.** Використання GPOS може вимагати значних ресурсів на планування, управління пам'яттю та фонові задачі.
- **Зазвичай потрібен мікропроцесор (і зовнішня ОЗП + ППЗП).**
- Розробник має **менше контролю над апаратурою** через використання додаткових рівнів абстракції. Переносимість коду на високому рівні.

Код та програма

Код і програма дуже тісно пов'язані. Ці терміни часто використовуються взаємозамінно для опису програмного забезпечення, яке керує вбудованим обладнанням, але деякі відмінності все ж таки є:

Код – це текстове представлення набору інструкцій або операцій, що написане на мові програмування і визначає логіку і поведінку програми. Наприклад, послідовність інструкцій для зчитування та обробки сигналів з різних датчиків, керування виведенням на дисплеї, взаємодії з іншими пристроями та багато іншого.

Програма – це результат написання коду, який включає весь код, драйвери, бібліотеки та інші компоненти, які необхідні для правильної роботи пристрою або системи. Програма

вбудованої системи зазвичай розробляється для виконання конкретного завдання або набору завдань, які визначають функціональність пристрою.

Існує кілька видів коду:

Вихідний код (Source Code) – це текст написаний програмістом на одній з мов програмування (C++, Java, Python та ін). Такий код легко можна прочитати та зрозуміти людині.

Об'єктний код (Object Code) – це результат перетворення вихідного коду у формат зрозумілий обчислювальному пристрою. Він створюється компілятором або іншими інструментами, які перетворюють текст зрозумілий людина у набір байтових кодів. Такий код потребує додаткової обробки для виконання на конкретній платформі.

Машинний код (Machine Code) – це набір нулів і одиниць, який представляє собою найбільш базові інструкції, які може розуміти центральний процесор конкретної архітектури. Це найнижчий рівень представлення програми.

Як мікроконтролер розуміє програміста?

Послідовність розуміння мікроконтролером програміста і виконання того, що він бажає, включає:

1. **Написання вихідного коду програмістом.** Висловлення думок програміста у вигляді тесту написаного на певній мові програмування.
2. **Компіляція**. Компілятор перетворює вихідний код спочатку в об'єктний код, а потім використовуючи інший інструмент, наприклад, лінкер, об'єднує об'єктний код в виконавчий код або бінарний виконуваний файл.
3. **Завантаження в пам'ять**. Виконуваний файл (програма) зберігається в пам'яті мікроконтролера (часто в області Flash або ROM).
4. **Запуск та ініціалізація**. Під час запуску мікроконтролера відбуваються різні етапи ініціалізації для підготовки його до нормальної роботи.
5. **Виклик коду.** Мікроконтролер починає виконання коду з певної адреси. Це може бути початкова адреса програми або адреса вектору переривань, залежно від конкретної архітектури.
6. **Виконання інструкцій.** Мікроконтролер читає інструкції з пам'яті, виконуючи їх послідовно. Кожна інструкція має своє конкретне завдання. Цикл виконання триває доки мікроконтролер увімкнений.
7. **Взаємодія з периферійними пристроями.** Мікроконтролер може взаємодіяти з різними периферійними пристроями (наприклад, введення-виведення, таймери, комунікаційні інтерфейси), читаючи та записуючи значення в їх регістри.

Інструкції мікроконтролера

Оскільки, код (програма) – це набір інструкцій, написаних на мові програмування, то доцільно розібратись, що таке інструкція.

Інструкція мікроконтролера – це основна операція, яку виконує процесор мікроконтролера. Кожна інструкція представляє собою команду, для виконання певної дії, такої як: завантаження даних з пам'яті, арифметичні обчислення, управління вводом-виводом або перехід до іншої частини програми. Кожен мікроконтролер має свій власний набір інструкцій, який визначається його архітектурою.

Щоб дізнатися про набір інструкцій для конкретного мікроконтролера, можна скористатися документацією, яку надає виробник. Така документація часто називається «регістро-орієнтованою документацією» або «посібником з програмування». Її можна знайти в таких місцях:

- **Офіційний веб-сайт виробника.** Там зазвичай присутні технічні описи, посібники та інша документація.
- **Довідкові матеріали та посібники виробника.** Ці матеріали можуть включати описи інструкцій, приклади коду та іншу корисну інформацію.
- **Форуми та спільноти розробників**, такі як Stack Overflow або форуми виробників, можуть бути корисними для отримання відгуків від інших розробників, які працюють з тим самим мікроконтролером.
- **Інструменти розробки.** Інтегровані середовища розробки (IDE) та інші інструменти розробки також можуть містити документацію та приклади коду для конкретних мікроконтролерів.

Наприклад, для роботи з мікроконтролерами STM32 від STMicroelectronics, на офіційному веб-сайті STMicroelectronics в розділі Documentation [1] можна знайти документацію, посібники та приклади коду. З прикладом «посібника з програмування» (для STM32 Cortex®-M0+) можна ознайомитись за посиланням [2].

Цикл виконання інструкцій (Fetch-Decode-Execute)

Цикл Fetch-Decode-Execute (зчитування, декодування, виконання) є базовим процесом в архітектурі процесора. Цей цикл визначає виконання інструкцій, які забезпечують обчислення та управління в програмах.

- **Fetch (зчитування).** Процесор отримує інструкцію з пам'яті програм, використовуючи адресу з регістру лічильника інструкцій (Program Counter). Після чого Program Counter збільшується, щоб вказувати на наступну інструкцію.
- **Decode (декодування).** Отримана інструкція (у вигляді машинного коду) розкодовується, щоб визначити, які дії (операції) повинні бути виконані мікроконтролером та над якими даними. Аналізується опкод (операційний код) або інші поля інструкції, які вказують на операцію та параметри, що її стосуються, режими адресації та робочі регістри. Наприклад, опкод може вказувати на операцію додавання, а додаткові біти можуть вказувати на регістри або операнди, які необхідно додати.
- **Execute (виконання).** Виконується фактична операція, зазначена розкодованою інструкцією. Використовуються наявні ресурси, такі як арифметично-логічні одиниці, регістри, оперативна пам'ять тощо. Результат може бути збережений у регістрах або в пам'яті.

Після завершення ітерації циклу Fetch-Decode-Execute, виконується наступна інструкція, і процес повторюється. Завдяки цьому забезпечується послідовність дій для обробки програм.

Мови програмування

Для програмування вбудованих систем використовуються різні мови програмування в залежності від конкретних вимог та обмежень вбудованих пристроїв. Вибір мови програмування важливий, оскільки вона визначає зручність розробки, продуктивність та можливості оптимізації коду.

Всі мови програмування можна розділити на мови низького та високого рівня.

Мови низького рівня – це мови наближені до машинного коду та апаратного забезпечення комп'ютера. Програми на мовах низького рівня можуть бути прямо виконані апаратурою без значного перетворення. Приклади: машинний код (інструкції в двійковому вигляді), асемблер (використовується символьне представлення інструкцій, що значно спрощує їх розуміння людиною).

Мови високого рівня – це мови набагато ближчі до мови, зрозумілої людині, і дозволяють виражати ідеї на високому рівні абстракції. Вони використовують складні структури даних та операції, що спрощують розробку програм. Приклади: C, C++, C#, Java, Python.

Також слід зазначити, що майже всі високорівневі мови прорамування мають багато можливостей, які інколи можуть бути не потрібними і призводити до надмірного використання пам'яті та ресурсів.

Ось деякі з популярних мов програмування для вбудованих систем:

- **C та C++** відомі своєю ефективністю та можливістю безпосередньо взаємодіяти з апаратним рівнем. Контроль управління пам'яттю може бути як перевагою так і недоліком, через відсутність автоматичного сміттєзбірника можуть викликати проблеми та витоки. Ці мови дозволяють оптимізувати роботу програми під конкретну апаратну платформу, але можуть бути складними, особливо для початківців.
- **Python** є легким для вивчення та використання, що робить його відмінним вибором для розробки простих вбудованих систем. Він має багату екосистему та різноманітні бібліотеки, що полегшують розробку. З точки зору використання ресурсів, порівняно з мовами C або C++, Python може бути менш ефективним.
- **Java** не є найпоширенішим вибором для вбудованих систем, але її доцільно використовувати для написання ПЗ, яке можна легко переносити між різними платформами. Java має простий синтаксис та велику кількість бібліотек, що полегшує розробку. Такі програми можуть вимагати більше пам'яті.
- **Assembler** (мова асемблера) дозволяє повністю контролювати апаратуру та ресурси мікроконтролера. Програми, написані на асемблері, можуть займати менше пам'яті, але їх написання може бути складним та вимагати глибокого розуміння архітектури процесора.

Вибір мови програмування для вбудованих систем залежить від конкретних вимог, обмежень, рівня програміста та характеристик пристрою. Також в одній системі може бути використано декілька мов програмування, наприклад, реалізація критичних за ресурсами функцій може виконуватися на асемблері, а решта коду написана на більш високорівневій мові, C/C++, Python чи Java – для спрощення розробки та підтримки високого рівня абстракції.

Зазвичай інженери вибирають мови C/C++ через їх ефективність та можливість уникнути вивчення всіх деталей «асемблерних» архітектур, що дозволяє застосовувати код на різних платформах не прив'язуючись до конкретної архітектури.

Синтаксис C/C++

Синтаксис мови програмування – це набір правил, які визначають, яким чином можна писати програми на даній мові. Він визначає правильну структуру програми, правила для створення виразів, команд і конструкцій, а також правила форматування коду.

Основні елементи синтаксису включають:

- Типи даних та змінні.
- Оператори та вирази.
- Керуючі конструкції для контролю потоку програми.
- Функції (підпрограми).

Типи даних та змінні

Дані та змінні в мікроконтролерних програмах є важливими компонентами, які дозволяють програмістам працювати з інформацією та забезпечують коректне функціонування вбудованих систем. Вони використовуються для:

- Збереження інформації.
- Забезпечення роботи програми.
- Підтримки структурного програмування.
- Взаємодії з користувачем та зовнішніми пристроями.

Дані (Data) – це інформація, яка може бути оброблена або збережена. Можуть включати числа, символи, рядки тексту, зображення, вимірювання з сенсорів, тощо.

Змінні (Variables) – це іменовані області пам'яті, які використовуються для зберігання та маніпуляції даними під час виконання програми. Змінні отримують свої імена від програміста і можуть представляти різноманітні типи даних, такі як числа, рядки, булеві значення тощо. Значення змінних можуть змінюватись під час виконання програми, що дозволяє працювати з даними динамічно, на відміну від констант.

Прості типи даних (базові):

- Цілі числа (integers): **int**, **short**, **long**, **long long** (у C++), **signed** та **unsigned** варіанти.
- Дійсні числа (floating point numbers): **float**, **double** (дійсні числа з подвійною точністю).
- Символи (characters): **char** використовується для зберігання одного символу або малих цілих чисел.
- Логічний тип (boolean): **bool** (у C++) може мати лише два можливих значення: true або false.
- Рядки (strings): **string** (у C++) представляє рядок символів.

Модифікатори (наприклад, **short**, **long**, **signed**, **unsigned**) можна використовувати для зміни розміру та діапазону. Крім цього, наявність та розмір типів може змінюватися в залежності від платформи та компілятора.

Інші типи даних:

- Порожній тип (**void**) – це спеціальний тип, який вказує на відсутність типу.
- Покажчик (**pointer**) зберігає адресу пам'яті іншої змінної чи об'єкта.
- Масиви (**array**) – упорядковані колекції елементів одного типу.
- Структури (**structure**) – комплексні типи, що об'єднують кілька інших типів даних.
- Класи (**class**, у C++) містять дані та методи для роботи з ними.
- Об'єднання (**union**) дозволяють використовувати один і той же блок пам'яті для зберігання різних типів даних.
- Перерахування (**enum**) представляє набір іменованих констант.

Оператори

Оператор у програмуванні є елементом мови програмування, який виконує певну операцію або виконує конкретний тип завдання. Оператори визначають дії, які виконуються над змінними чи значеннями в програмі. Можна виділити декілька груп операторів:

- Арифметичні оператори (+, -, /, *, ++, --, %).
- Оператори порівняння (==, !=, <, >, <=, >=).
- Логічні оператори (&&, ||, !).
- Оператори присвоювання (=, +=, -=, *=, /=, %=).
- Побітові оператори (&, |, ^, ~, <<, >>).

Контроль потоку програми

Контроль потоку програми забезпечується конструкціями, які дозволяють визначати порядок виконання інструкцій. Основні засоби контролю потоку включають:

1. **Оператори умови:**
   - **`if`** – використовується для виконання блоку коду, якщо певна умова істинна.

```
if (умова)
{
    // код, який виконається, якщо умова істинна
}
```

   - **`else`** – використовується для визначення альтернативного блоку коду, який виконається, якщо умова в if невірна.

```
if (умова)
{
    // код, який виконається, якщо умова істинна
}
else
{
    // код, який виконається, якщо умова в if невірна
}
```

   - **`else if`** – дозволяє перевіряти додаткові умови, якщо попередні умови в if або else if були невірними.
   - **`switch`** – використовується для вибору одного з численних варіантів в залежності від значення виразу.

```
switch (вираз)
{
    case значення1:
        // код для значення1
        break;
    case значення2:
        // код для значення2
        break;
    // інші варіанти
    default:
        // код для випадку, якщо жодне значення не відповідає виразу
}
```

2. **Цикли:**
   - **`for`** – використовується для виконання блоку коду задану кількість разів.

```
for (ініціалізація; умова; крок)
{
    // код, який виконається поки умова істинна
}
```

   - **`while`** – виконує блок коду, доки задана умова істинна.

```
while (умова)
{
    // код, який виконається поки умова істинна
}
```

   - **`do-while`** – схожий на while, але гарантує виконання блоку коду принаймні один раз, оскільки перевірка умови відбувається після виконання блоку.

```
do
{
    // код, який виконається принаймні один раз
}
while (умова);
```

3. **Оператори переходу:**
   - **`break`** – використовується для завершення виконання циклу або виходу зі switch.
   - **`continue`** – призупиняє поточну ітерацію циклу і переходить до наступної.
   - **`return`** – повертає значення з функції та припиняє її виконання.

Ці конструкції дозволяють програмістам ефективно керувати виконанням програми.

Функції

Функція – це фрагмент програмного коду, який виконує певну задачу. Функції дозволяють розділити складну та об’ємну програму на простіші та менші, самостійні частини (підпрограми), що полегшує розробку, зрозумілість коду та можливість використання одного і того ж коду в різних місцях програми.

Основні аспекти функцій в C/C++:

1. **Оголошення функції** (Function Declaration) містить ім'я функції, тип повернення (тип даних, який функція повертає) та список параметрів, які функція приймає. Наприклад:

```
int add(int a, int b);
```

2. **Визначення функції** (Function Definition) містить код функції, включаючи визначення змінних, операції та будь-які інші інструкції, які функція виконує. Наприклад:

```
int add(int a, int b)
{
    return a + b;
}
```

3. **Виклик функції** (Function Call). Функцію можна викликати в іншому місці програми для виконання визначених нею операцій. Наприклад:

```
int result = add(3, 4);
```

4. **Параметри та аргументи**. Параметри – це значення, які функція приймає при виклику. Аргументи – це конкретні значення, передані функції.

```
void printMessage(char message[])
{
    printf("%s\n", message);
}

int main()
{
    printMessage("Hello, World!");
    return 0;
}
```

У цьому прикладі, **`message`** – це параметр функції **`printMessage`**, а текст **`"Hello, World!"`** – це аргумент, переданий при виклику.

5. **Повернення значення.** Функції можуть повертати значення за допомогою ключового слова **`return`**. Тип значення, що буде повернуто, повинен відповідати типу повернення, визначеному в оголошенні функції.

```
int square(int x)
{
    return x * x;
}
```

Функція **`square`** повертає квадрат переданого їй значення `x`.

Покажчики

Покажчики (Pointers) – це особливий тип змінних в багатьох мовах програмування, що зберігають адресу пам'яті іншої змінної чи об'єкта. Вони дозволяють прямий доступ і маніпулювання об'єктами в пам'яті та надають потужний інструмент для роботи з динамічною пам'яттю.

Основні концепції пов'язані з покажчиками включають:

1. **Оголошення покажчиків:**
```
int *ptr;  // Покажчик на ціле число
```
2. **Отримання адреси змінної:**
```
int variable = 10;
int *ptr = &variable;  // ptr тепер містить адресу змінної variable
```
3. **Розіменування покажчика:**
```
int value = *ptr;  // value тепер містить значення змінної, на яку вказує ptr
```
4. **Арифметика покажчика** (покажчики можна збільшувати або зменшувати на певну кількість байт за допомогою арифметичних операцій):
```
ptr++;  // Збільшує адресу на одиницю (вказує на наступний елемент)
ptr--;  // Зменшує адресу на одиницю (вказує на попередній елемент)
```
5. **Масиви** (ім'я масиву є покажчиком на його перший елемент):
```
int array[5] = {1, 2, 3, 4, 5};
int *ptr = array;  // Покажчик на початок масиву
int thirdElement = *(ptr + 2);  // Отримання третього елемента
```
6. **Динамічне виділення пам'яті:**
```
// Динамічне виділення пам'яті для масиву з 5-ти цілих чисел
int *dynamicArray = (int*)malloc(5 * sizeof(int));
```
7. **Звільнення динамічно виділеної пам'яті:**
```
free(dynamicArray);  // Звільнення пам'яті після використання
```

Покажчики дозволяють працювати з динамічною пам'яттю, структурами даних та функціями, які приймають аргументи через покажчики. Вони є важливим інструментом для оптимізації та ефективного використання ресурсів в програмах.

Точка входу в програму

В мові програмування C/C++ точкою входу в програму є функція **`main()`**. Після запуску програми, першою виконається саме ця функція. Синтаксис функції **`main()`** виглядає так:

```
int main()
{
    // Код програми
    return 0;  // Опціональний оператор повернення
}
```

Функція **`main()`** може бути визначена без аргументів або з ними, наприклад так:

```
int main(int argc, char *argv[])
{
    // Код програми з аргументами командного рядка
    return 0;
}
```

Також, варто зазначити, що **`main()`** може повертати значення, яке буде інтерпретовано операційною системою як «код стану виходу» програми.

Структура проекту

Структура проекту може варіюватися залежно від конкретних потреб і організаційних вимог. Проте, деякі загальні концепції включають наступні елементи:

```
project/
├── src/
│   ├── main.c
│   ├── module1.c
│   └── module2.c
└── include/
    ├── module1.h
    └── module2.h
```

- **src/** або **source/** – директорія, де зазвичай розміщуються вихідні файли проекту. У таких файлах міститься реалізація функцій та логіка програмного коду.
- **include/** або **header/ –** директорія, де розміщують файли заголовків (header files). Зазвичай, у цих файлах оголошуються інтерфейси функцій, структур даних і констант, які будуть використовуватися у вихідних файлах.
- **build/** або **bin/** – директорія, де можуть зберігатися виконуючі файли, бінарні файли та інші тимчасові файли, що створюються під час компіляції.
- **Makefile** або **CMakeLists.txt** – файл, який містить інструкції для системи збірки (наприклад, Make або CMake), яка дозволяє автоматизувати процес компіляції та збірки проекту.

Програмні модулі та бібліотеки (розподіл коду на декілька файлів)

Розподілення коду на файли допомагає організувати та структурувати програму. Зазвичай, код розділяють на декілька файлів, де кожен файл має свою відповідальність. Великі програми розділяються на головний файл (наприклад, main.c), файл(и) з визначеннями функцій та файли з визначенням структур чи інших типів даних.

Наприклад, якщо в проекті є файли main.c, functions.c та functions.h, то вони можуть мати наступний вміст:

- **main.c**:

```c
#include "functions.h"

int main()
{
    myFunction();  // Виклик функції з functions.c
    return 0;
}
```

- **functions.c**:

```c
#include "functions.h"
#include <stdio.h>
void myFunction()
{
    printf("Hello from myFunction!\n");
}
```

- **functions.h**:

```
// Прототип функції
void myFunction();
```

Включення файлу з розширенням «**.c**» (наприклад, **`#include "functions.c"`**) в інший файл чи програму не є обов'язковим. Це може привести до проблем, подвійного включення коду в кількох місцях та викликати конфлікти і помилки компіляції.

Зазвичай в мові С код функцій розміщують у відповідних файлах з розширенням .c, а прототипи функцій та інші оголошення розміщують у файлах заголовків з розширенням .h. Файли .h потім включають у файли .c, в яких необхідно використовувати функції чи структури з інших файлів.

Такий підхід дозволяє розділити інтерфейс (оголошення функцій) та реалізацію (код функцій) та використовувати їх в інших частинах програми, забезпечуючи чистоту та підтримку коду.

Вирази **`#include <file>`** та **`#include "file"`** використовуються для включення файлів у програму, але вони мають різні особливості.

- **`#include <file>`** (загальні заголовкові файли) – вказує компілятору де шукати файл в стандартних системних шляхах. Зазвичай використовується для включення загальних заголовкових файлів, які постачаються разом із компілятором або операційною системою.
- **`#include "file"`** (локальні заголовкові файли) – вказує компілятору де шукати файл в поточному каталозі програми або в інших вказаних місцях. Зазвичай використовується для включення власних заголовкових файлів, створених користувачем для конкретного проекту.

В наведеному вище прикладі коду, **`#include <stdio.h>`** – це директива препроцесора С, яка вказує компілятору підключити (вставити) вихідний код, що міститься у стандартному заголовковому файлі **`stdio.h`**, який файл містить оголошення функцій для введення/виведення, таких як **`printf`** та **`scanf`**, а також деякі інші засоби для роботи зі стандартним введенням/виведенням.

При створенні програмного забезпечення для комунікації, сервісів безпеки, управління транзакціями і т.д. доцільно використовувати існуючі бібліотеки стеків протоколів, так зване проміжне програмне забезпечення – Middleware.

Middleware – це програмне забезпечення або шар програм, який додається між апаратним та програмним забезпеченням для полегшення роботи з різними пристроями та сервісами. Middleware використовується для спрощення взаємодії між різними компонентами системи, роблячи програмування та використання апаратних можливостей більш зручним та ефективним, дозволяючи розробникам концентруватися на функціоналіді своїх додатків. Приклад Middleware – офіційні та сторонні бібліотеки Arduino; засоби операційних систем та інше.

Процес збірки програми

Процес збірки (Building) – це сукупність кроків, що включає компіляцію, лінкування, та інші операції, які необхідні для перетворення вихідного коду в виконуваний (двійковий) файл.

Можна виділити декілька основних кроків:

1. **Попередня обробка (Preprocessing).** Препроцесор обробляє вихідний код програми перед компіляцією.
   - Вхідними файлами для цього етапу є файли із розширенням *.c/*.h (Source Files).
   - Результатом цього етапу є препроцесовані файли з розширенням **.i**.

   Попередня обробка починається з видалення коментарів із вихідного *.c файла. Далі виконується проходження по коду для обробки директив попередньої обробки, включаючи розгортання макросів, підключення файлів, обробку всіх спеціальних інструкцій, які починаються з символу #.

2. **Компіляція (Compilation).** На цьому етапі попередньо оброблений вихідний код конвертується в асемблерний код, який специфічний для різних архітектур процесорів. Відбувається не пряме відображення рядків коду, а скоріше декомпозиція операцій C/C++ на численні операції асемблера.
   - Вхідними файлами для цього етапу є препроцесовані файли *.i.
   - Результатом цього етапу є файли з асемблерним кодом *.s або *.asm.

   Компіляцію можна розділити на декілька основних етапів:
   - Аналіз синтаксису та семантики програми.
   - Розпізнавання токенів, таких як ключові слова, ідентифікатори, оператори та літерали.
   - Перевірка, чи токени організовані згідно правил мови C/C++, щоб уникнути синтаксичних помилок.
   - Перевірка, чи речення, яке було розібрано, має правильне значення (чи відсутні семантичні помилки).
   - Відстеження змінних у структурі, відомій як символьна таблиця.
   - Трансляція внутрішнього представлення. Перетворення конструкцій мови високого рівня в асемблерну форму.
3. **Асемблювання (Assembler)**. На цьому етапі код асемблера, який був згенерований компілятором, конвертується в об'єктний код.
   - Вхідними файлами є *.asm або *.s.
   - Результатом цього етапу є файли *.o або *.obj – об'єктний код.

   Сучасні компілятори можуть генерувати об'єктний код без використання окремого асемблера.

4. **Лінкування (Linking) та розміщення.** Об'єктні файли, створені на попередньому етапі, потрібно зв'язати між собою, а також, за потреби, додати код з бібліотек. Лінкер об'єднує ці файли та створює виконуваний файл, який потім можна розміщувати в пам'яті мікроконтролера. На цьому етапі також визначаються адреси фізичної пам'яті та можливість переміщення програми.

Прикладом «лінкування» є написання великої книги, розділеної на різні розділи. Після закінчення написання, потрібно зібрати всі розділи разом та розмістити їх в правильному порядку, щоб кожна сторінка була на своєму місці. Аналогічно і з програмами. При написанні програми, вона може бути розділена на різні частини. Лінкер (спеціальна програма) бере всі ці частини і розміщує їх в правильному порядку в пам'яті. Кожна частина отримує свою власну "адресу", яка вказує на місце, де ця частина знаходиться в пам'яті. Таким чином, розміщення – це визначення того, де саме в пам'яті знаходяться різні частини програми, щоб вони могли правильно працювати під час виконання.

Також можуть застосовуватись різні утиліти, наприкад Make. **Make** – це утиліта, яка автоматизує процес компіляції та збірки програмного коду. Вона дозволяє визначити, як саме із декількох вихідних файлів створити виконуваний файл, і визначає порядок виконання операцій для цього. **Make-файл** – це текстовий файл, який містить інструкції, що вказують на залежності між різними файлами проекту і команди для їх компіляції та лінкування.

Середовище розробки

Середовище розробки (IDE, Integrated Development Environment) – це програмний комплекс, який надає розробникам усі необхідні інструменти для роботи над проектами. Таке середовище об'єднує різні інструменти, які сприяють зручному написанню, відлагодженню і тестуванню коду, а саме:

- **Текстовий редактор** – місце де розробник може створювати та редагувати код.
- **Компілятор** – забезпечує перетворення вихідного коду в машинний.
- **Налагоджувач –** інструмент для виявлення та виправлення помилок.
- **Система керування версіями** – дозволяє відстежувати та контролювати зміни в коді, щоб спростити спільну роботу декількох розробників над проектом.
- **Система автоматизації збірки** (Build Automation) – допомагає автоматизувати процес збірки програми з різних компонентів.
- **Система управління проектом** – надає можливість організувати та керувати структурою проекту (включати файли, бібліотеки, залежності і т. д.).
- **Додаткові інструменти** для тестування, аналізу коду, профілювання та іншого.

Інтеграція всіх цих компонентів в єдиному середовищі дозволяє розробникам зосередитися на написанні високоякісного коду, спрощуючи багато інших процесів.

Toolchain

Toolchain – це набір конкретних варіантів інструментів (компілятори, асемблери, лінкери та інше), які використовуються для створення програмного забезпечення. Для різних платформ чи архітектур вони можуть значно відрізнятися. Нижче наведено деякі популярні інструменти:

1. Середовища розробки:
   - Arduino IDE – просте та легке у використанні середовище розробки, спеціально створене для програмування Arduino-сумісних платформ.
   - Visual Studio та Visual Studio Code – середовища від Microsoft, які підтримують різні мови програмування та платформи.
   - Eclipse – зручне IDE, яке можна налаштовувати за допомогою розширень та плагінів. Підтримує велику кількість мов програмування та платформ розробки.

- Keil – популярне IDE для мікроконтролерів з ядрами ARM.
- Arm Development Studio – IDE від ARM, призначене для роботи з їхніми процесорами.
- Code Composer Studio – середовище розробки від Texas Instruments.

2. Компілятори:

- GCC (GNU Compiler Collection) – відомий вільно розповсюджуваний компілятор, який підтримує широкий спектр архітектур.
- ARM Compiler – розроблений спеціально для ARM.

Існує багато інших компіляторів для різних платформ.

3. Засоби відладки:

- GDB (GNU Debugger) – потужний інструмент для відладки на рівні машинного коду. Дозволяє відстежувати виконання програми, перевіряти значення змінних, виконувати покроково та багато іншого.
- Segger J-Link – програмно-апаратний засіб для відладки та емуляції.
- OpenOCD (Open On-Chip Debugger) – відкрите програмне забезпечення для відладки.

Наприклад, для мікроконтролерів ATmega328, які використовуються у платформі Arduino Uno, може застосовуватись такий Toolchain:

1) **Arduino IDE** з вбудованим текстовим редактором – для написання програми.
2) **AVR-GCC** – інструментарій, що включає препроцесор, компілятор, асемблер та лінкер.
3) **AVRDude** – утиліта, що дозволяє завантажувати програмне забезпечення на мікроконтролери AVR з використанням різних методів програмування та комунікаційних інтерфейсів.

Запитання для самоперевірки

1. Що в себе включає розробка програмного забезпечення для вбудованих систем?
2. Які різновиди програмування вбудованих систем існують?
3. Яка роль коду та програми в контексті вбудованих систем?
4. Як мікроконтролер розуміє програміста?
5. Які мови програмування можуть бути використані для вбудованих систем?
6. Як проходить цикл виконання інструкцій мікроконтролера (Fetch-Decode-Execute)?
7. Як відбувається взаємодія з пам'яттю в мікроконтролерах?
8. Які бувають оператори в мові програмування C/C++?
9. Як здійснюється контроль потоку програми?
10. Як визначаються та використовуються функції?
11. Що таке покажчики і яку роль вони виконують?
12. Що таке точка входу в програму та як її визначити?
13. Як відбувається розподіл коду на декілька файлів?
14. Як відбувається процес збірки програми для вбудованих систем?
15. Що включає середовище розробки для вбудованих систем і які існують набори для розробки?
16. Яка документація є необхідною для програмування вбудованих систем?

Тести

1. Які ключові етапи розробки ПЗ для вбудованих систем?
   A) Аналіз вимог, проектування, реалізація, тестування.
   B) Проектування, тестування, випробування, супровід.
   C) Аналіз вимог, проектування, реалізація.
   E) Аналіз вимог, реалізація, випробування.
2. Чим відрізняються мови низького та високого рівня?
   A) Мови низького рівня – це мови програмування для вбудованих систем, а мови високого рівня – це мови для веб-програмування.
   B) Мови низького рівня – це мови програмування для мікроконтролерів, а мови високого рівня – це мови для настільних програм.
   C) Мови низького рівня – це мови, які близькі до машинного коду, а мови високого рівня – це мови, які більш абстрактні та зручні для програмістів.
   D) Мови низького рівня – це мови, які використовують початківці, а мови високого рівня – це мови для досвідчених програмістів.
3. Назвіть особливості машинної мови та мови асемблера.
   A) Мова асемблера – це мова, яка складна для розуміння, а машинна мова – це мова, яка покращує читабельність коду.
   B) Машинна мова – це мова, що використовується для написання програм для машин, а мова асемблера – це мова, що використовується для програмування мікроконтролерів.
   C) Мова асемблера – це набір символів, які повністю відображають машинний код. Машинна мова – це набір бітів, що прямо виконується процесором.
   D) Машинна мова – це мова програмування високого рівня, а мова асемблера - це мова програмування низького рівня.
4. В чому особливості використання мов високого рівня?
   A) Вони дозволяють працювати з апаратурою безпосередньо.
   B) Вони мають велику ефективність в роботі з об'єктами в оперативній пам'яті.
   C) Вони є більш абстрактними і полегшують процес програмування.
   D) Вони мають більш високий рівень безпеки.
5. З якими типами даних можливо працювати за допомогою мови C/C++?
   A) Тільки з цілими числами від -2 147 483 648 до 2 147 483 647.
   B) З числами з плаваючою крапкою та цілими числами.
   C) З числами, символами, рядками та масивами.
   D) З символами, рядками та масивами.
6. Чим відрізняється створення константи від змінної?
   A) Константа може змінюватися під час виконання програми, а змінна не може.
   B) Константа має постійне значення, а змінна посилання на значення.
   C) Константа не може змінюватися під час виконання програми, а змінна може.
   D) В C/C++ константи та змінні - це одне і те саме.

7. Які види операторів існують в C/C++?
   A) Строкові, числові, символьні, вказівникові.
   B) Арифметичні, логічні, порівняння, присвоєння, побітові.
   C) Умовні, циклічні, виразові, масивні.
   D) Функціональні, методичні, агрегаційні, вказівні.
8. Що таке контроль потоку програми?
   A) Покроковий вивід результатів програми, реалізується за допомогою команди відлагодження.
   B) Механізм керування послідовністю виконання інструкцій, реалізується за допомогою умовних конструкцій та циклів.
   C) Автоматизація виконання операцій, реалізується за допомогою створення функцій та методів.
   D) Зміна структури програми під час виконання, реалізується за допомогою маніпуляцій з пам'яттю.
9. Що таке процес збірки програми?
   A) Перетворення вихідного коду в виконуваний файл.
   B) Створення вихідного коду з готових бібліотек.
   C) Перевірка програми на наявність помилок.
   D) Перетворення двійкового коду в вихідний файл.
10. Які інструменти необхідні для перетворення вихідного коду в машинний?
    A) Компілятор, лінкер.
    B) Редактор коду, відлагоджувач.
    C) Інтерпретатор, віртуальна машина.
    D) Контроль версій, система тестування.
11. Назвіть різновиди програмування вбудованих систем.
    A) Програмування мобільних додатків, програмування вбудованих систем.
    B) Програмування на апаратному рівні, програмування під операційну систему.
    C) Web-програмування, програмування на Java, мобільне програмування.
    D) Програмування ігор, робототехніка, інтелектуальні системи.
12. Що таке надлишкові програмні витрати та від чого вони залежать?
    A) Надмірне використання програмного забезпечення, залежність від типу операційної системи.
    B) Надмірна залежність від програмного забезпечення.
    C) Зайва або непотрібна витрата ресурсів, залежність від реалізації програмного коду та його оптимізації.
    D) Перевищення обсягу виконуваних операцій, залежність від користувацького інтерфейсу.

13. Які особливості програмування без операційної системи?
    A) Програма працює безпосередньо з апаратним обладнанням та має низьку переносимість.
    B) Програма взаємодіє з операційною системою, що відповідає за розподіл ресурсів і виконання запитів.
    C) Програма взаємодіє з додатковим програмним забезпеченням, а не з операційною системою.
    D) Програма виконується на сервері, який не має операційної системи.
14. Особливості програмування з ОС:
    A) Програма взаємодіє з додатковим апаратним забезпеченням.
    B) Програма виконується на операційній системі сервера.
    C) Програма використовує сервіси операційної системи.
    D) Операційна система взаємодіє з користувачем.
15. Що таке код та програма?
    A) Код - це послідовність символів, програма - це набір функцій.
    B) Код - це програма на мові асемблера, програма - це програмний продукт, що виконує певну дію.
    C) Код - це текст, що містить програмні інструкції, програма - це вищезазначений код та інші компоненти для виконання конкретної функції.
    D) Код - це текстовий файл, що містить програму, програма - це алгоритмічний опис процесу виконання задачі.
16. Назвіть декілька видів коду:
    A) Машинний код, вихідний код, об'єктний код.
    B) Об'єктний код, програмний код, вхідний код.
    C) Асемблерний код, машинний код, збірковий код.
    D) Вхідний код, вихідний кодкод, виконуваний код.
17. Що таке інструкції мікроконтролера та операційні коди?
    A) Це команди для роботи з операційною системою, що виконуються на мікроконтролері.
    B) Інструкції - це команди, що виконуються процесором. Вони представлені у вигляді операційних кодів.
    C) Інструкції мікроконтролера - це команди, що виконуються на мікроконтролері. Операційні коди - це коди помилок, що повертаються при виконанні команд.
    D) Інструкції - це набір документів, що містить операційні коди до певного сімейства процесорів.
18. Цикл Fetch-Decode-Execute – це:
    A) Процес послідовного виконання команд на мікроконтролері.
    B) Процес перевірки та виконання команд в операційній системі.
    C) Процес взаємодії зовнішніх пристроїв з мікроконтролером.
    D) Процес паралельного виконання команд на мікроконтролері.

ТЕМА 3. ВНУТРІШНЬОСИСТЕМНА ТА МІЖСИСТЕМНА КОМУНІКАЦІЯ

План лекції

- Зв'язок і його види
- Типи передачі даних (послідовний, паралельний)
- Методи послідовного зв'язку (синхронний, асинхронний)
- Інтерфейси та комунікаційні протоколи (UART, SPI, I2C)

Текстова частина лекції

Зв’язок і його види

Зв'язок – це процес передачі даних між двома або більше пристроями чи системами. Взаємодія мікроконтролерів із зовнішнім світом та іншими пристроями грає важливу роль у функціонуванні будь-якої вбудованої системи.

Можна виділити три види передачі даних:

- **Simplex** – передача відбувається тільки в одному напрямку. Один пристрій може лише передавати дані, а інший може лише приймати. Наприклад, телевізор, який може тільки отримувати сигнали від телевежі, тобто телевежа виступає в ролі лише передавача (Transmitter, TX), а телевізор – лише приймача (Reciever, RX).
- **Half-duplex** – передача даних відбувається в обох напрямках, але не одночасно. Наприклад, при розмові по рації потрібно говорити та слухати по черзі. В цьому випадку кожен пристрій має як TX, так і RX, але вони по черзі використовують одну лінію зв’язку.
- **Duplex** – у цьому режимі передача даних відбувається в обох напрямках одночасно, тобто обидва пристрої можуть одночасно передавати та приймати дані. Наприклад, телефонна розмова, де обидва абоненти можуть одночасно говорити та слухати. В цьому випадку кожен пристрій також має TX та RX, але вони з’єднуються за допомогою окремих ліній зв’язку.

Паралельна та послідовна (серійна) комунікація

Комунікація між мікросхемами/пристроями може відбуватися за допомогою різних методів, але основні дві концепції – це паралельна та послідовна (Serial, серійна). Обидві використовуються в різних випадках і мають свої переваги та недоліки.

1. **Паралельна комунікація:**
    - **Декілька бітів передаються одночасно**. За рахунок цього швидкість передачі даних може бути дуже високою.
    - **Використання багатьох каналів**. Кількість фізичних каналів (провідники або шини) залежить від кількості біт для одночасної передачі. Це може значно підвищити складність виконання та вартість реалізації.
    - **Обмежена дальність**. При використанні паралельних провідників кількість перехресних електромагнітних перешкод зростає з зростанням довжини з’єднання.

- **Застосування**: шини даних (напр. ранні версії стандарту PCI), міжпроцесорний обмін даними.

2. **Послідовна комунікація:**
   - **Один біт за раз**. Інформація передається послідовно, по одному біту за раз.
   - **Використання одного каналу для передачі даних** (наприклад, один провід або один оптично-волоконний кабель).
   - **Велика дальність**. Менше імовінрість втратити дані через електромагнітні завади, що дозволяє передавати дані на більші відстані.
   - **Застосування**: взаємодія з переферійними пристроями, давачами, а також між вбудованими системами.

Обираючи між серійною та паралельною комунікацією, важливо враховувати конкретні потреби проекту, витрати ресурсів, відстань передачі та інші параметри. Завдяки своїм перевагам кожен метод має унікальні застосування.

Важливо, що послідовна комунікація поділяється на синхронну та асинхронну.

Синхронна послідовна комунікація

При синхронній комунікації відправник та отримувач синхронізують свої дії за допомогою спільного тактового сигналу. Це означає, що дані передаються визначеною послідовністю бітів, і кожен біт відправляється в певний момент часу. Забезпечується надійна передача даних, оскільки кожен біт гарантовано буде отриманий в потрібний момент часу.

Основні етапи синхронногої комунікації:

1) Передавач повідомляє приймач про намір передачі даних.
2) Дані передаються у визначені ітервали часу (синхронізовані за тактовим сигналом).
3) Після завершення передачі, передавач надсилає закриваючий біт.
4) Приймач, отримавши дані, надсилає підтвердження про це. Доки передавач не отримає підтвердження він повторюватиме передачу.

Особливості:

- Передавач та приймач використовують спільний сигнал синхронізації.
- Кількість даних, яку можна передати за один раз, може варіюватися, від декількох бітів до кілобайт чи навіть більше.

Асинхронна послідовна комунікація

Асинхронна комунікація, з іншого боку, не вимагає спільного тактового сигналу. Вона є більш гнучною, але налаштування між передавачем та отримувачем повинні бути узгоджені, а саме:

- **Швидкість передачі** (Baud rate). Бод (Baud) — це одиниця виміру швидкості передачі даних у телекомунікаціях та електроніці, що визначає кількість символів, які передаються за одну секунду.
- **Кількість бітів даних в одному «пакеті»**. Зазвичай використовується від 5 до 9 бітів.
- **Визначення стартового та стопового бітів**, а також використання біту парності (для виявлення помилок передачі).

Основні етапи асинхронногої комунікації:

1) Передавач починає передачу, відправляючи стартовий біт, який зазвичай має значення "0". Він сигналізує приймачу про початок нового пакета даних.
2) Передавач відправляє дані (наприклад, 8 бітів на символ).
3) Якщо використовується біт парності – він додається після бітів даних.
4) Передавач відправляє стоповий біт (зазвичай один, інколи декілька), який має значення "1". Він сигналізує приймачу про завершення передачі.

Особливості:

- Замість спільного сигналу синхронізації, для визначення місця закінчення одного біту і початку наступного, передавач і приймач використовують внутрішні сигнали.
- Для виділення корисного навантаження використовуються стартові та стопові біти.

Інтерфейси в вбудованих системах

Інтерфейс – механізм, за допомогою якого різні компоненти, пристрої чи системи можуть взаємодіяти один з одним, включаючи обмін даними, командами та іншою інформацією.

Інтерфейси можуть бути внутрішніми та зовнішніми:

Внутрішні інтерфейси – використовуються для взаємодії між різними модулями чи компонентами в межах одного пристрою чи мікроконтролера. Це можуть бути внутрішні шини, регістри та інші механізми.

Зовнішні інтерфейси – використовуються для взаємодії з іншими системами або пристроями. Наприклад, для введення та виведення даних, обробки сигналів, взаємодії через різні порти та інше.

Основні типи інтерфейсів включають:

1. Фізичні інтерфейси – роз'єми та порти, тобто фізичні конектори, які дозволяють підключати пристрої до системи (USB-порти, HDMI-роз'єми, роз'єми живлення, тощо), а також засоби передачі даних (різні кабелі, шини).
2. Логічні інтерфейси:
   - Комунікаційні протоколи – визначають правила обміну даними між пристроями з'єднаними на рівні апаратного забезпечення (I2C, SPI, UART).
   - Інтерфейси API (Application Programming Interface) – логічні інтерфейси програмного забезпечення, які визначають, як програми можуть взаємодіяти одна з одною (бібліотеки, класи, веб-служби тощо).
3. Системні інтерфейси – дозволяють взаємодіяти з рівнем операційної системи, включаючи виклики системи, роботу з ресурсами та керування процесами.
4. Мережеві інтерфейси – визначають, як пристрої можуть взаємодіяти в мережевому середовищі. Наприклад, між рівнями мережевої моделі TCP/IP.
5. Інтерфейси для обміну даними – формати обміну даними, що визначають структуру та правила представлення даних (JSON, XML та ін.).
6. Інтерфейси користувача:
   - Графічні інтерфейси користувача (GUI) – забезпечують взаємодію користувача з системою через графічні елементи, такі як меню, вікна тощо.
   - Інтерфейси командного рядка – дозволяють користувачеві вводити команди та отримувати відповіді в текстовому вигляді.

Топології зв’язку

У вбудованих системах розповсюджено декілька топологій, які визначають спосіб організації з'єднання між різними компонентами системи, основні з них включають:

- **P2P (Point-to-Point, Peer-to-Peer)** – топологія в якій кожен компонент підключений безпосередньо до іншого компонента, утворюючи з'єднання "один до одного". Це забезпечує прямий шлях для обміну даними. P2P часто використовується для сполучення окремих сенсорів або пристроїв з головним контролером.
- **Загальна шина (Bus)** – топологія в якій всі компоненти підключені до одного загального каналу передачі даних, який називається шиною. Кожен компонент може передавати та приймати дані через цей канал. Ця топологія проста у розширенні, але вона може мати обмеження з пропускною здатністю.
- **Зірка (Star)** – топологія в якій всі компоненти підключені до центрального вузла (мікроконтролера). Всі дані спочатку надходять йому, а він направляє їх до необхідних приймачів. Така топологія є простою в управлінні, але, якщо центральний вузол вийде з ладу, зв’язок між рештою компонентів зникне.

Комунікаційні протоколи

Комунікаційний протокол – це набір правил, які визначають формат і порядок обміну даними між пристроями або системами, що дозволяє пристроям ефективно взаємодіяти між собою. Комунікаційні протоколи можуть визначати способи кодування даних, формати пакетів, процедури розпізнавання помилок, механізми керування потоком та передачею даних і багато іншого.

Протоколи у вбудованих системах можна розділити на дві групи:

1. **Внутрішньосистемні протоколи**, які призначені для обміну даними між різними компонентами або модулями в межах однієї вбудованої системи (друкованої плати). Вони зазвичай працюють на низькому рівні апаратного забезпечення, такому як шина даних або шина керування, і дозволяють мікросхемам взаємодіяти між собою. Наприклад: I2C, SPI, CAN.
2. **Міжсистемні протоколи** використовуються для обміну даними між різними системами, що можуть бути фізично віддалені. Наприклад: USB, UART, USART.

UART

UART (Universal Asynchronous Receiver/Transmitter) – це апаратний модуль, який використовується для забезпечення послідовної асинхронної комунікації між двома пристроями (топологія P2P). Протоколи передачі даних через UART є одним з найпростіших, найстаріших та найпоширеніших.

Особливості UART:

- **Мінімум 2 дроти.** По одному дані передаються (TX – передавач), а по іншому приймаються (RX – приймач). Для керування потоком можуть застосовуватись, ще два, для сигналів RTS (Request to Send) та CTS (Clear to Send).
- **Серійна комунікація.** Дані передаються у блоках (зазвичай по 8 біт), по одному біту за раз.

- **Асинхронний режим.** Тобто, без спільного тактового сигналу. Замість нього, для синхронізації, дані супроводжуються стартовим та стоповим бітами.
- **Стандарти обміну даними.** UART використовується різними стандартами: RS-232, EIA-232, TIA-232, RS-485, COM port, інфрачервоний зв’язок.
- **Простота та широке застосування.** Через свою простоту та широку сумісність, UART використовується для забезпечення зв'язку між різноманітними пристроями. Він є надійним інструментом, особливо в випадках, де необхідно швидко та ефективно передавати невеликі об'єми інформації.
- **Швидкість передачі даних.** UART може працювати на різних швидкостях (бітова швидкість), що дозволяє адаптувати його до вимог конкретної системи.

Послідовність використання UART включає наступні кроки:

1) Перед початком комунікації потрібно налаштувати параметри UART на обох пристроях.
2) Підключити передавач (TX) одного пристрою до приймача (RX) іншого пристрою і навпаки, а також з'єднайти землю (GND) обох пристроїв.
3) У програмному забезпеченні, на кожному пристрої, відкрити UART-порт для передачі та прийому даних.
4) Відправити дані через UART, викликавши відповідний метод або функцію.
5) Обробити прийняті дані.
6) Після завершення комунікації можна закрийти відповідний порт, щоб звільнити ресурси і підготувати інтерфейс для інших операцій.

Відправка даних по UART складається з кількох ключових етапів:

1) Передавальний пристрій надсилає стартовий біт, який позначає початок передачі (лінія передачі приймає низький логічний рівень).
2) Далі передаються дані, які складаються з 8 бітів (може бути інша кількість, в залежності від налаштувань). Логічному «0» відповідає напруга 0 В, а логічній «1» відповідає 5 В.
3) Надсилається біт парності, ящко він використовується.
4) Завершує передачу даних надсилання стопового біту, що піднімає лінію передачі до високого рівня.

Для забезпечення додаткового часу для підготовки повільного пристрою до наступного кадру може використовуватись додатковий стоповий біт. В результаті, кадр, що містить 8 біт даних, в залежності від налаштувань, може займати максимум 12 біт, а мінімум – 10.

Можна виділити два підходи програмістів до застосування UART:

1) **Bit-banging** – передбачає керування виводами мікроконтролера замість використання апаратних UART модулів.

Переваги:

- Підходить для мікроконтролерів, які не мають вбудованого апаратного UART.
- Гнучкість у визначенні параметрів зв'язку, таких як швидкість передачі, формат кадру та контроль помилок.

Недоліки:

- Зазвичай менш ефективний і повільніший, ніж апаратний UART.

2) **Периферійні пристрої UART**. У більшості мікроконтролерів існують вбудовані апаратні модулі UART, які забезпечують апаратне управління передачею та отриманням даних через відповідний UART-порт. Ці модулі можуть мати високу швидкість передачі даних та підтримку різних параметрів зв'язку.

Переваги:

- Використання апаратних ресурсів пристрою UART знижує навантаження на процесор.
- Висока швидкість передачі даних і ефективність.

Недоліки:

- Залежність від наявності апаратного UART модуля, а також їх кількості та типів.

В склад кожного модуля UART входить передавальний та приймальний регістр, кожен з яких під'єднаний до власного регістру зсуву. Обидва регістри зсуву синхронізуються з тактовим сигналом пристрою.

Процес передачі даних розпочинається з їх завантаження у передавальний регістр. З нього дані поміщаються в регістр зсуву, звідки вони відправляються (зсуваються) по одному біту до приймача (іншого пристрою).

Приймач отримує дані через регістр зсуву, по одному біту. Далі вони переміщаються в приймальний регістр. Таким чином, дані переміщуються від передавача до приймача через послідовні тактові імпульси, забезпечуючи надійну передачу і отримання інформації.

Приклад:

В залежності від обраного мікроконтролера доцільно використовувати відповідні бібліотеки для роботи з UART. Для мікроконтролерів сумісних з Arduino, це може бути бібліотека SoftwareSerial [3] або вбудований клас Serial, що містить методи для передачі та отримання даних через UART-порт.

Основні методи класу Serial [4]:

- **`begin`**(*speed*) або **`begin`**(*speed*, *config*) – ініціалізує UART-з'єднання з заданою швидкістю передачі даних (бодами). Параметр *speed* визначає швидкість передавання даних у бодах (9600, 115200 тощо), а *config* – вказує на різноманітні додаткові налаштування, такі як розмір даних, біти перевірки парності та стоп-біти.
- **`print`**(*value*) або **`print`**(*value*, *format*) – дозволяють надсилати дані через UART. Параметр *value* – це значення, яке потрібно передати. Це може бути будь-яке ціле число, десяткове число або рядок. Перед надсиланням дані автоматично конвертуються в рядок. Необов'язковий параметр *format* вказує на формат виведення значення, наприклад: DEC для десяткового формату, HEX – шістнадцяткового, BIN – двійкового. Якщо замість **`print`** використовувати **`println`**, то до кінця виводу буде додано символи переходу на наступний рядок «**`\r\n`**».
- **`available()`** – повертає кількість доступних для читання символів в буфері вводу.
- **`read()`** – читає символ з буфера вводу (якщо він є) і повертає його код або -1, якщо буфер порожній.

Детальніше про ці та інші методи класу Serial можна дізнатись з офіційної документації [4].

Зразок програми для передачі та отримання даних по UART виглядає наступним чином:

```
      char UARTbyte;
void setup()
      {
        // Встановлення швидкості передачі даних
  Serial.begin(9600);

  // Відправлення даних через UART
  Serial.println("Hello, world!");
}

void loop()
{
  // Перевірка наявності отриманих даних
  if (Serial.available() > 0)
  {
    // Читання отриманих даних
    UARTbyte = Serial.read();
  }
}
```

У функції **setup** встановлено швидкість передачі даних 9600 біт на секунду, на якій відправляється рядок **"Hello, world!"**. У головному циклі програми **loop** перевіряється наявність отриманих даних і, якщо вони є, відбувається їх читання, за допомогою **Serial.read()**, та збереження в змінній **UARTbyte**.

USART (Universal Synchronous Asynchronous Receiver Transmitter) – це апаратний модуль, який поєднує у собі можливості асинхронного і синхронного зв'язку. Він подібний до UART, але додатково має можливість працювати у синхронному режимі та на більш високих швидкостях.

COM-порт (RS-232)

COM-порт (Communication Port) – це інтерфейс для зовнішнього з'єднання комп'ютера або іншого пристрою з периферійними пристроями, такими як принтери, миші, клавіатури тощо. В операційній системі кожен COM-порт ідентифікується унікальним номером (наприклад, COM1, COM2, COM3). Цей інтерфейс відповідає стандарту RS-232.

RS-232 (Recommended Standard 232 [5]) – стандарт для серійної асинхронної передачі даних між пристроями. Він визначає електричні та механічні властивості інтерфейсу. Оригінально був розроблений для підключення терміналів до модемів і з часом став популярним стандартом для з'єднань між різними електронними пристроями.

Основні характеристики RS-232:

- **Напруга сигналу.** Логічному «0» відповідає позитивна напруга (від +3 до +15 В), а логічній «1» – негативна (від −3 до −15 В).
- **Сигнальні лінії.** Включає сигнальні лінії RTS (Request to Send) та CTS (Clear to Send), для керування передачею та прийомом даних.

SPI

SPI (Serial Peripheral Interface) – це послідовний синхронний протокол передачі даних між двома або більше мікроконтролерами, сенсорами, дисплеями, модулями пам'яті та іншими периферійними пристроями.

Особливості SPI:

- **Топологія «майстер-слейв».** Лише один пристрій виступає в ролі головного, а інші йому підпорядковуються.
- **Синхронізація.** В SPI синхронізація базується на тактовому сигналі (SCLK) і використанні ребер цього сигналу для визначення часу передачі бітів.
- **Мінімум три лінії:** MOSI (Master Out Slave In), MISO (Master In Slave Out), SCLK (Serial Clock). Додатково, при роботі з багатьма пристроями, до кожного з них йде окрема лінія Chip Select (CS або SS).
- **Окремі лінії для передачі та прийому даних.** Можливість повного дуплексу значно підвищує швидкість передачі даних.
- **Немає адресації пристроїв.** «Майстер» взаємодіє з конкретним «слейвом» за допомогою сигналу Chip Select по відповідній лінії.
- **Неперервна передача даних.** Відсутність стартових та стопових бітів дозволяє передавати дані безперервно.
- **Відсутність підтвердження доставки та перевірки помилок.**

Послідовність передачі даних по SPI включає наступні кроки:

1) «Майстер» генерує тактовий сигнал для синхроннізації.
2) «Майстер» вибирає чіп, з яким хоче взаємодіяти, встановлюючи низький рівень на відповідному виводі CS (Chip Select).
3) «Майстер» відправляє дані по одному біту через вивід MOSI (Master Out Slave In), а «слейв» їх отримує.
4) За потреби «слейв» може відправляти відповідь через вивід MISO (Master In Slave Out).
5) «Майстер» встановлює на виводі CS високий рівень, що вимикає обмін даними з вибраним «слейвом».

Приклад програми для комунікації по SPI:

```
#include <SPI.h>

#define CS1 10
#define CS2 11


/*  NLSF595 MOSI - SI
     SCLK - SCK
       SS - RCK  */

void setup()
{
  pinMode(CS1, OUTPUT);
  pinMode(CS2, OUTPUT);
```

```
  //Ініціалізація SPI
  SPI.beginTransaction(SPISettings(14000000, LSBFIRST, SPI_MODE0));
}

void loop()
{

  // Chip 1 – встановлення кольору RGB LED
  digitalWrite(CS1, LOW);
  SPI.transfer(0b00100100);
  digitalWrite(CS1, HIGH);
  delay(1000);


  // Chip 2 – встановлення кольору RGB LED
  digitalWrite(CS2, LOW);
  SPI.transfer(0b00100100);
  digitalWrite(CS2, HIGH);
  delay(1000);

}
```

У цьому прикладі для взаємодії мікроконтролера з драйверами RGB-світлодіодів використовується бібліотека SPI.h [6].

На початку визначаються піни, які відповідають за лінії вибору чіпу CS. В даному випадку, це піни 10 та 11 для ліній **CS1** та **CS2**, відповідно.

В функції **`setup`** ініціалізується SPI. Метод **`beginTransaction`** встановлює параметри зв'язку. Він приймає об'єкт **`SPISettings`**, який містить три параметри: швидкість передачі даних (*speedMaximum*), порядок передачі даних (*dataOrder*) та режим зв'язку (*dataMode*).

- *speedMaximum* – визначає максимальну швидкість передачі даних в Гц. У наведеному прикладі використовується 14 000 000, тобто 14 МГц.
- *dataOrder* – визначає який біт має передаватися першим: найбільш значущий біт (MSB) або найменш значущий біт (LSB). У наведеному прикладі використовується **`LSBFIRST`**.
- *dataMode* – вказує на стосіб передачі даних. **`SPI_MODE0`** – це один з чотирьох режимів у якому зчитування даних відбувається на передньому фронті (зростаючому краю) тактового імпульсу, а передача даних – на задньому фронті (спадному краю).

Далі, в **`loop`**, за допомогою функції **`digitalWrite(CS1, LOW)`** обирається чіп з яким потрібно комунікувати. Метод **`transfer`**, передає байт даних **`0b00100100`** цьому пристрою, а після передачі пін CS1 піднімається до рівня HIGH за допомогою **`digitalWrite(CS1, HIGH)`**. Потім, після паузи в одну секунду, обирається другий чіп та надсилається аналогічний байт даних вже йому.

I2C

I2C (Inter-Integrated Circuit) – це протокол послідовної синхронної комунікації через два провідника – лінію даних «SDA» та лінію синхронізації (тактову) «SCL». I2C широко використовується для зв’язку між ІМС, що розташовані на одній платі, а також для зв'язку з різними сенсорами, дисплеями та іншими периферійними пристроями.

Особливості I2C:

- **Half-duplex, загальна «шина».** Дані передаються по єдиній лінії даних «SDA», до якої підключаються всі мікросхеми.
- **Багатомайстерний протокол.** Є можливість підключення більше одного майстра до шини, але одночасно активно спілкуватись може лише один з них.
- **Адресація.** Кожен пристрій повинен мати унікальну адресу, щоб «майстер» міг вказувати з ким він хоче спілкуватися.
- **Умови початку та завершення.** Комунікація починається коли на лінії SDA відбувається перехід з високого до низького рівеня напруги, при наявності високого рівня на лінії SCL. Умовою завершення є перехід з низького до високого рівня напруги спочатку на лінії SCL, а потім на SDA, після чого рівень SCL залишається високим.
- **Структуровані повідомлення.** Кожне повідомлення включає: умову початку; кадр адреси (для ідентифікації приймача); біт читання/запису; біт підтвердження/відмови від підтвердження; один або декілька кадрів з корисним навантаженням; умову завершення.
- **Режими передачі даних.** Режими зчитування і запису визначають чи є мікросхема отримувачем/передавачем даних.
- **Швидкість передачі даних.** Три основних режими роботи: стандартний (100 кбіт/с), швидкий (400 кбіт/с) та високошвидкісний (3,4 Мбіт/с). Комунікація зазвичай повільніша ніж по SPI та залежить від завантаження шини.
- **Підтягуючі резистори.** Виходи SDA та SCL потребують підтягуючих резисторів, для уникнення проблем з синхронізацією.

Послідовність передачі даних по I2C включає наступні кроки:

1) При високому рівні на лінії SCL «майстер», перемикає лінію SDA від високого до низького рівня, що відповідає умові початку початку передачі.
2) «Майстер» надсилає адресу «слейва» по лінії SDA, яка може займати 7 або 10 біт.
3) Далі надсилається біт читання/запису, який вказує чи «майстер» надсилає дані до «слейва» (0) чи отримує їх (1).
4) «Майстер» очікує від «слейва» біт підтвердження/відмови (ACK/NACK). Якщо «слейв» успішно отримав адресу, він відправляє ACK; якщо не вдається розпізнати адресу або є помилка, «слейв» надсилає NACK.
5) «Майстер» надсилає або отримує дані від «слейва», де кожен байт даних потребує окремого підтвердження.
6) Після передачі/отримання даних «майстер» генерує сигнал, що відповідає умові завершення, спочатку встановлюючи на лінії SCL, а потім на SDA високий рівень напруги.

Приклад програми для комунікації по I2C:

```
#include <Wire.h>     // Бібліотека для роботи з шиною I2C

#define MPU_addr 0x68 // Адреса давача MPU6050

int AccX,AccY,AccH,Temp;      // Змінні для зберігання прискорення та температури

void setup()
{
  Serial.begin(9600);  // Відкриття монітору послідовного порту
  Wire.begin();        // Ініціалізація шини I2C
  Wire.beginTransmission(MPU_addr);// Початок передачі за вказаною адресою
  Wire.write(0x6B);    /* Перший байт даних (номер регістру MPU6050,
                          що відповідає за живлення та налаштування)*/
  Wire.write(0);       // Другий байт даних (нулі = запуск давача)
  Wire.endTransmission(true); // Завершення передачі та звільнення лінії
}

void loop()
{
  Wire.beginTransmission(MPU_addr); // Почати надсилання даних з вказаною адресою
  Wire.write(0x3B);                /* Надсилання байту даних
                                      (адреса регістру для зчитування з давача)*/
  Wire.endTransmission(false);      // Завершення надсилання без відключення лінії
  Wire.requestFrom(MPU_addr,8,true);/* Отримання 8 байтів даних від давача
                                      та відключення лінії*/
  AccX=Wire.read()<<8|Wire.read();  // Зчитування 16 біт з прискоренням по X
  AccY=Wire.read()<<8|Wire.read();  // Зчитування 16 біт з прискоренням по Y
  AccH=Wire.read()<<8|Wire.read();  // Зчитування 16 біт з прискоренням по H
  Temp=Wire.read()<<8|Wire.read();  // Зчитування 16 біт з значенням температури

  // Конвертація та вивід отриманої інформації у монітор послідовного порту
  Serial.print("Прискорення по X = "); Serial.print(AccX/ 16384.0);
  Serial.print(" | по Y = "); Serial.print(AccY/ 16384.0);
  Serial.print(" | по H = "); Serial.print(AccH/ 16384.0);
  Serial.print(" | Температура = "); Serial.println(Temp/340.00+36.53);

  delay(1000); // Затримка перед наступним повідомленням
}
```

В даному прикладі виконується зчитування даних з давача MPU6050 за допомогою шини I2C. Програма ініціалізує шину I2C, налаштовує давач, а потім зчитує дані про прискорення (по X, Y, H) та температуру. Отримані дані конвертуються та виводяться у монітор послідовного порту для подальшого аналізу.

Для взаємодії по I2C використовується бібліотека Wire.h [7]. Основні методи якої включають **`begin`** для ініціалізації шини, **`beginTransmission`** та **`endTransmission`** для початку та завершення передачі даних, **`write`** для надсилання даних, **`requestFrom`** для отримання даних, та **`read`** для читання отриманих даних. Детальніше про ці та інші методи класу **`Wire`** можна дізнатись з офіційної документації [7].

UART vs SPI vs I2C

Вибір протоколів залежить від конкретних вимог системи, таких як кількість підключених пристроїв, необхідна швидкість передачі даних та інші параметри. Ключові відмінності I2C, SPI та UART:

Кількість ліній:

- UART – дві лінії основних лінії (TX – передача, RX – прийом) та, за потреби, ще дві додаткових (RTS та CTS, для контролю потоку).
- SPI – три основних лінії, для з'єднання двох мікросхем (MOSI, MISO, SCK). При вібльшій кількості використовуються додаткові лінії (CS/SS) до кожної з мікросхем.
- I2C – лише дві лінії (SDA, SCL).

Топологія:

- UART – один до одного (P2P).
- SPI – шинна топологія з одним «майстром» та багатьма «слейвами».
- I2C – шинна топологія, що підтримує наявність декількох «майстрів».

Адресація:

- UART – не має механізму адресації пристроїв. Кожен пристрій використовує свій унікальний порт.
- SPI – механізм адресації не використовується. Вибір пристрою відбувається за допомогою сигналу по окремій лінії Chip Select (CS) для кожної мікросхеми.
- I2C – використовується механізм адресації, а адреса отримувача включається в структуру кожного повідомлення.

Режими роботи:

- UART – асинхронний (не потрібен спільний тактовий сигнал); повний дуплекс (дозволяє передавати та отримувати дані одночасно); кожен байт супроводжується стартовим та стоповим бітами.
- SPI – синхронний; повний дуплекс; синхронізація здійснюється за допомогою спільного тактового сигналу (SCK), який визначає час передачі та отримання кожного біту.
- I2C – синхронний, напівдуплекс; синхронізація здійснюється за допомогою тактової лінії SCL (Serial Clock); використовуються стартові та стопові умови.

Швидкість передачі даних (точні значення можуть змінюватися в залежності від конкретних реалізацій пристроїв та умов експлуатації):

- UART – найнижча швидкість порівняно з SPI та I2C, яка значно залежить від моделі модуля та довжини кабеля (орієнтовно десятки або сотні кбіт/с).
- SPI – найбільша швидкість, через відсутність потреби у передачі адрес, стартових та стопових бітів (одиниці та десятки Мбіт/с).
- I2C – декілька режимів роботи: 100 кбіт/с, 400 кбіт/с, 3,4 Мбіт/с, 5 Мбіт/с.

Споживана енергія: I2C та SPI – зазвичай вимагають більше енергії (порівняно з UART) через більш складний обмін даними.

Використання:

- UART зазвичай використовується в простих застосунках, для зв'язку між пристроями на середній дальності. Наприклад: мікроконтролери, давачі, принтери, комп’ютери, Bluetooth і GSM модулі тощо.
- SPI використовується на невеликих відстанях для високошвидкісного обміну даними між мікроконтролерами та периферійними пристроями. Наприклад: для зв'язку з пам'яттю, АЦП, сенсорами, дисплеями тощо.
- I2C підходить для систем, де багато мікросхем повинні спілкуватися між собою через одну шину.

Протоколи бездротової комунікації

Безпровідна комунікація в вбудованих системах грає важливу роль, дозволяючи пристроям обмінюватися даними без фізичного з'єднання.

Основні протоколи безпровідної комунікації включають:

- **Bluetooth –** це один із найпопулярніших протоколів для короткодіючого бездротового зв'язку. Використовується для з'єднання різних пристроїв, таких як смартфони, навушники, клавіатури, миші та інше. Різні класи Bluetooth мають різні діапазони роботи, від сантиметрів до десятків метрів.
- **Wi-Fi** – це стандарт для бездротового з'єднання в мережах локального доступу (LAN). Використовується для підключення комп'ютерів, смартфонів, планшетів, камер та інших пристроїв до Інтернету або локальних мереж. Має велику пропускну здатність, але може споживати значну кількість енергії.
- **Zigbee** – протокол, що розроблений для взаємодії між різними пристроями в "розумних" будинках, промислових системах та інших вбудованих застосуваннях. Зазвичай використовується для створення мереж малого радіусу з дуже низьким споживанням енергії.
- **RFID** (Radio-Frequency Identification) використовується для безконтактного зчитування інформації з тегів або карток у системах контролю доступу, при інвентаризації та в інших областях.
- **NFC** (Near Field Communication) є стандартом для обміну даними на невеликих відстанях (зазвичай декілька сантиметрів). Використовується в мобільних платежах, обміні контактами, квитках та інших додатках.
- **LoRa** (Long Range) – це протокол для бездротового зв'язку на великі відстані з низькою потужністю. Застосовується для розумних міст, сільського господарства та інших сценаріїв, де потрібно покрити значу площу.
- **5G** – новітній стандарт мобільного зв'язку, забезпечує високу швидкість передачі даних, низьку затримку та підтримку великої кількості підключених пристроїв. Має потенціал для різних вбудованих застосувань, включаючи Інтернет речей (IoT).

Кожен з цих протоколів має свої унікальні властивості та підходить для різних вбудованих застосувань залежно від вимог до швидкості, дальності, споживання енергії та інших факторів.

Запитання для самоперевірки

1. Які види зв'язку існують, і яка їх відмінність?
2. Що таке Simplex, Half-duplex та Duplex?
3. Назвідь типи передачі даних, і особливості кожного з них?
4. Які методи послідовного зв'язку ви знаєте?
5. В чому основні відмінності синхронного та асинхронного зв'язку?
6. Що означає швидкість передачі в бодах і як вона впливає на комунікацію?
7. Що таке комунікаційний протокол і яке його призначення?
8. Що таке UART і як він використовується для передачі даних?
9. Що таке Bit-banging і в чому його особливості?
10. Які переваги та недоліки використання UART?
11. Що таке SPI і які лінії з'єднання використовуються для нього?
12. Як відбувається вибір чіпа в контексті SPI?
13. Які особоливості підключення по протоколу I2C?
14. Які можливості передачі даних пропонує I2C, в чому його основні переваги?

Тести

1. В чому особливість Duplex-ної передачі даних?
   A) Передача даних лише в одному напрямку одночасно.
   B) Передача даних в обох напрямках одночасно.
   C) Дублююча передача даних в одному напрямку по черзі.
   D) Передача даних в обох напрямках по черзі.
2. Що означає SPI?
   A) Синхронний паралельний інтерфейс.
   B) Послідовно-паралельний інтерфейс.
   C) Стандартний послідовний інтерфейс.
   D) Послідовний переферійний інтервейс.
3. Що таке "Швидкість передачі в Бодах"?
   A) Кількість байтів, що передаються за один такт.
   B) Кількість символів, що передаються за одну секунду.
   C) Кількість годин, необхідних для передачі одного байта.
   D) Кількість бітів, які можна передати за один тактовий імпульс.
4. Що таке Chip Select і яке його призначення в SPI?
   A) Мікросхема для вибору швидкості передачі даних.
   B) Сигнал, який вказує пристрою, що він повинен обмінюватись даними.
   C) Чіп, що визначає адресу пристрою в SPI мережі.
   D) Програмний код для вибору потрібного пристрою в мережі.
5. Що таке I2C?
   A) Інтерфейс паралельного зв'язку між 2-ма пристроями.
   B) Інтерфейс для взаємодії 2-х контролерів.
   C) Інтерфейс зворотного зв'язку між контролером та периферійними пристроями.
   D) Ссинхронний протокол зв'язку на двох дротах.

6. Що таке UART?
   A) Універсальний адаптивний роз'єм телекомунікацій.
   B) Універсальний асинхронний приймач-передавач.
   C) Універсальний апаратний послідовний передавач.
   D) Універсальний асинхронний паралельний протокол.
7. Що таке комунікаційний протокол?
   A) Апаратний компонент для перетворення даних у сигнали.
   B) Стандартний алгоритм для обробки даних.
   C) Типовий спосіб з'єднання пристроїв у мережі.
   D) Набір правил для обміну даними між пристроями.
8. Що характеризує метод Bit-banging?
   A) Використання бітів для створення шуму на лінії зв'язку.
   B) Програмне керування виводами мікроконтролера для передачі або прийому даних.
   C) Використання шумоподібних сигналів для зниження спектральної щільності потужності.
   D) Перетворення бітів у шум і навпаки.
9. Який принцип роботи лежить в основі асинхронного зв'язку?
   A) Використання зовнішнього годинника для синхронізації передачі.
   B) Використання стартових та стопових бітів для сигналізації початку та кінця передачі.
   C) Використання бездротового зв'язку між пристроями для передачі даних.
   D) Підтримка оператором стану з'єднання для організації передачі даних.
10. Які основні переваги використання I2C?
   A) Лише 2 провідника, не потрібен загальний тактовий сигнал.
   B) Використовує лише два дроти, підтримує декількох майстрів та декількох слейвів.
   C) Немає стартових та стопових бітів, тому дані можуть передаватися безперервно.
   D) Розмір кадру даних не обмежений 8 бітами.
11. Які основні переваги використання SPI?
   A) Лише 2 провідника, не потрібен загальний тактовий сигнал.
   B) Використовує лише два дроти, підтримує декількох майстрів та декількох слейвів.
   C) Немає стартових та стопових бітів, тому дані можуть передаватися безперервно.
   D) Краща обробка помилок.
12. Які основні переваги використання UART?
   A) Лише 2 провідника, не потрібен загальний тактовий сигнал.
   B) Використовує лише два дроти, підтримує декількох майстрів та декількох слейвів.
   C) Немає стартових та стопових бітів, тому дані можуть передаватися безперервно.
   D) Вища швидкість передачі даних, ніж у I2C (майже вдвічі).
13. Які типи передачі даних існують?
   A) Перпендикулярний та паралельний.
   B) Паралельний та змінний.
   C) Асинхронний та синхронний.
   D) Паралельний та послідовний.

ТЕМА 4. ІНТЕРНЕТ РЕЧЕЙ

План лекції

- Мережа – це основа
- Особливості IoT
- Архітектура IoT
- Складові IoT
- Типи взаємодії в IoT
- Рівнева мережева архітектура (модель)
- Популярні протоколи

Текстова частина лекції

Мережа – це основа

Мережеві технології визначають засоби, протоколи та методи, які використовуються для передачі даних між вбудованими системами.

Існують такі типи мереж:

- **Персональна мережа (PAN, Personal Area Network)** – охоплює невелику територію, яка зазвичай використовується однією особою. Наприклад, мережа між смартфоном та ПК, на основі Bluetooth.
- **Локальна мережа (LAN, Local Area Network)** – об'єднує пристрої в межах обмеженої географічної області, такої як будинок, офіс чи кампус.
- **Глобальна мережа (WAN, Wide Area Network) –** охоплює великі географічні відстані та зазвичай з'єднує багато LAN. Приклажом такої мережі є Інтернет, який є ключовим елементом Інтернету речей, оскільки забезпечує обмін даними між системами розташованими на величезній території.

Мережі кожного з наведених типів можуть бути:

- **Дротовими**, в яких використовується фізичний кабель для передачі інформації між пристроями (наприклад, Ethernet або USB). Надійність та швидкість передачі даних таких мереж зазвичай вища.
- **Бездротовими (Wireless)**, в яких передача даних відбувається за допомогою радіохвиль або інфрачервоного випромінювання. Прикладами є Wi-Fi, мобільні (3G, 4G, 5G) та супутникові мережі.

Всі ці мережі надають можливість обміну інформацією між різноманітними пристроями, наприклад:

- комп’ютерами та ноутбуками;
- смартфонами та планшетами;
- смарт-телевізорами та мережевими принтерами;
- системами відеоспростереження та акустичними системами.

Це лише декілька прикладів. Різноманітність пристроїв постійно зростає та відображає зростання розумних технологій вцілому.

Інтернет речей (Internet of Things, IoT) – це концепція, яка описує мережеве підключення фізичних об'єктів (речей) до глобальної мережі (Інтернету), що, по суті, створює «розумні» об'єкти, які з будь-якої точки світу (де є Інтернет) можуть взаємодіяти один з одним або з користувачем. Коротко можна описати IoT, як «вбудовані системи, що підключені до Інтернету».

Основні ідеї:

- Об'єкти із світу фізичних речей підключаються до Інтернету, що дозволяє їм збирати та обмінюватися даними, а також отримувати команди від інших пристроїв чи користувачів.
- Кожна річ в системі має унікальний ідентифікатор, що дозволяє однозначно ідентифікувати її у мережі.
- Крім речей, в систему входять пристрої, що відповідають за моніторинг, контроль та керування речами.

Відмінності між персональними комп'ютерами та вбудованими системами підключеними до Інтернету:

- Функціональність та застосування.
- Розмір та форм-фактор.
- Обчислювальна потужність та енергоспоживання.
- З’єднання та комунікація.

Тенденції у впровадженні IoT

Кількість розумних пристроїв вже перевищила кількість людей на планеті та продовжує зростати. Інтернет речей стає актуальним для багатьох галузей, включаючи промисловість, містобудування, транспорт, охорону здоров'я, сільське господарство, домашню автоматизацію та інші. З розвитком технологій потреби ринку постійно змінюються, але можна виділити деякі ключові тенденції:

- Зростання застосувань у промисловості (Industrial IoT, IIoT) для оптимізації виробничих процесів, підвищення ефективності обладнання, покращення управління та моніторингу стану пристроїв.
- Розвиток розумних міст, покращення інфраструктури, включаючи управління транспортом, енергозбереження, моніторинг лічильників, контроль якості повітря та води, управління відходами та багато іншого.
- Розвиток домашньої автоматизації для зручного та ефективного управління освітленням, опаленням, кондиціонуванням повітря, системами безпеки, побутовою технікою та іншими пристроями за допомогою смартфонів або голосових асистентів.
- Модернізація сільського господарства допомагає керувати ресурсами, контролювати стан рослин та тварин, автоматизувати процеси поливу, розподілу добрив та збору врожаю.

- Автономні транспортні засоби та розвиток їх систем моніторингу і навігації, дистанційного керування і оновлення, а також взаємодії з іншим транспортом та об’єктами.
- Впровадження IoT у сфері охорони здоров’я для моніторингу здоров’я в реальному часі, надання консультацій та нагадувань, підключення обладнання та автоматизації процесів обміну даними.
- Більше уваги приділяється кібербезпеці. Компанії та організації шукають способи захисту своїх пристроїв і даних від кібератак.
- Для забезпечення сумісності між різними пристроями та платформами, розвиваються стандарти та протоколи.
- Росте значення аналітики даних. Для отримання цінної інформації, прогнозування тенденцій та прийняття рішень впроваджується засоби на основі «штучного інтелекту» (AIoT – Artificial Intelligence IoT).

Рівнева мережева архітектура (модель)

Рівнева мережева архітектура – це концепція, яка описує організацію та функціональність різних рівнів або прошарків комунікаційної системи, які взаємодіють між собою для забезпечення комунікації в мережі, структурованого підходу до проектування та розуміння мережевих систем.

Основні принципи рівневої архітектури:

- **Розділення функцій** – кожен рівень виконує конкретні функції та завдання, ізольовані від інших рівнів. Розділення функцій дозволяє легко модифікувати або покращувати окремі компоненти мережевого стеку.
- **Розподіл завдань між рівнями** – кожен рівень відповідає за конкретні аспекти комунікації, такі як передача даних, обробка помилок, адресація тощо. Різні рівні можуть бути імплементовані та оптимізовані незалежно один від одного.
- **Взаємодія через інтерфейси** – кожен рівень надає послуги для вищих рівнів через чіткі інтерфейси. Інтерфейси дозволяють взаємодіяти з різними рівнями без необхідності дослідження деталей їхньої реалізації.
- **Модульність та розширюваність** – дозволяє впроваджувати нові технології без значних змін в існуючій архітектурі.

Існує кілька мережевих моделей, які визначають структуру та функції мережевих архітектур. Дві найвідоміші моделі – це модель **OSI (Open System Interconnection Reference Model)** та модель **TCP/IP (Transmission Control Protocol/Internet Protocol)**.

Модель OSI складається з семи рівнів, кожен з яких відповідає за певну функціональність, таку як фізична передача даних, маршрутизація, керування сесією та інші. Де-факто, взаємодію в Інтернеті описує мережева модель TCP/IP, яка складається з чотирьох рівнів:

1. **Рівень доступу до мережі (Link Layer)** – відповідає за фізичне підключення пристроїв до мережі (Ethernet, Wi-Fi).
2. **Рівень мережі (Network Layer)** – відповідає за маршрутизацію даних через мережу. Протокол IP (Internet Protocol) є ключовим на цьому рівні.

3. **Транспортний рівень (Transport Layer)** – відповідає за передачу даних відсортованих рівнем Застосунків. Протоколи TCP та UDP.
4. **Рівень застосунків (Application Layer)** – відповідає за забезпечення служб для програм та користувачів. Сюди входять такі протоколи, як: HTTP/S, MQTT, XMPP та інші, спеціально розроблені для Інтернету речей.

Архітектура IoT

Найпростішу модель архітектури IoT можна описати за допомогою трьох рівнів:

- **Сприйняття/Відчуття (Perception/Sensing)** – це фізичний рівень, який включає сенсори та пристрої, які здатні сприймати/відчувати фізичні параметри оточуючого середовища, такі як температура, вологість, освітленість тощо. Сенсори перетворюють ці параметри на цифрові дані, які потім можна передавати по мережі.
- **Мережевий рівень (Network)** відповідає за підключення до інших розумних речей, мережевих пристроїв і серверів та транспортування даних між ними.
- **Прикладний/Додатків (Application)** – це рівень на якому дані отримані від серсорів обробляються та аналізуються, на основі чого відбувається прийняття рішень. Він відповідає за надання користувачам специфічних програмних послуг.

Зібрані з давачів дані зазвичай проходять через такі області:

1) **Гранична область (Edge) –** це найближче місце розташування обчислювальних ресурсів однієї організації (серверів, сховищ даних, маршрутизаторів тощо) до мережі іншої організації. Відповідає за:
   - Взаємодію з "речами" (давачами та актуаторами).
   - Інтеграцію з рівнем обробки подій через мережу Internet.
2) **Область обробки подій та аналітики** – це сукупність різноманітних обчислювальних ресурсів об’єднаних мережею Інтернет. Відповідає за:
   - Адміністрування та керування пристроями з граничної області.
   - Обробку подій з використанням аналітичних сервісів.
   - Виконання хмарних обчислень.
   - Зберігання, обробку та направлення даних потрібним додаткам.
   - Машинне навчання для формування висновків про об'єкт моніторингу.
3) **Кінцеві застосунки –** це мобільні додатки, веб-інтерфейси та інші інструменти для контролю, адміністрування та взаємодії з системою IoT на стороні користувача.

Екосистема IoT

Основні елементи, що входять в екосистему IoT:

- **Речі (Things)** – це конкретні об'єкти, вбудовані системи, розумні пристрої, які можуть включати датчики (температури, вологості, руху, GPS), актуатори, засоби для обробки інформації тощо.
- **Комунікаційна мережа** – це інфраструктура, що забезпечує підключення речей до Інтернету бездротовим (наприклад, Wi-Fi, Bluetooth, Zigbee) або дротовим (Ethernet) способом. Вона включає мережеві пристрої та обладнання (маршрутизатори, комутатори, хаби, мости, мережеві карти та інше).

- **Сервери та хмарні сервіси (Cloud Services)**. Дані можуть передаватися на приватні сервери або до хмарних сервісів для подальшого збереження, обробки та аналізу. Також хмарні сервіси пропонують готові інструменти для віддаленого керування.
- **Протоколи комунікації.** Для взаємодії речей та передачі даних використовуються різні протоколи, як базові (стек TCP/IP) так і спеціально розроблені для IoT.
- **Програмне забезпечення**, яке: дозволяє речам працювати у мережі; виконує обробку зібраних даних, їх аналіз, для виявлення трендів, прогнозування відмов та прийняття рішень; забезпечує захисту даних від несанкціонованого доступу використовуючи шифрування, автентифікацію та інші методи підвищення безпеки.

Типи взаємодії в IoT

Існує три основні типи взаємодії в Інтернеті речей:

1. **Thing-to-Human (T2H)** – речі надсилають інформацію або повідомлення користувачам. Наприклад: сповіщення від системи безпеки про виявлення руху вдома; сповіщення від "розумного" холодильника про відсутність продуктів або виявлені несправності.
2. **Human-to-Thing (H2T)** – цей тип взаємодії передбачає надсилання людиною команд до речей для виконання певної дії. Наприклад, керування освітленням, термостатами, системами безпеки за допомогою мобільного додатка або голосових команд..
3. **Thing-to-Thing (T2T)** або **Machine-to-Machine (M2M)** – взаємодія між речами без прямої участі користувача. Наприклад, системи "розумних" будинків, де різні пристрої автоматично реагують на зміни в оточенні. Датчики світла вмикають освітлення, коли виявляють недостатній рівень освітленості та інше.

Крім цього, можна зустріти наступні варіанти взаємодії: Secure Thing-to-Thing, Thing-to-Server та Thing-to-Cloud.

Підключення вбудованих систем до Інтернету

Можна виділити наступні варіанти підключення вбудованих систем до мережі Інтернет:

1) Пряме підключення до Інтернету:
   - Дротова мережа Ethernet.
   - Бездротова мережа Wi-Fi.
2) Підключення через додаткові шлюзи:
   - Bluetooth – речі підключаються до локального шлюзу, який з'єднується з Інтернетом. Наприклад, смартфон як шлюз для фітнес-браслета.
   - Zigbee – пристрої підключаються до спеціального координатора.
   - LoRaWAN – потребується LoRa-шлюз.
3) Підключення через мобільні мережі (2G/3G/4G/5G, NB-IoT), за допомогою відповідних модемів.
4) Підключення через електричні лінії (PLC, Power Line Communication).

Ethernet

Ethernet – один з найпоширеніших протоколів для провідних мереж, що використовується для передачі фреймів даних в локальних мережах. Вбудовані системи можуть використовувати Ethernet для підключення до мережі. Для цього знадобиться наступне:

- **Ethernet-контролер або Ethernet-модуль.** Це може бути частина самого MCU або окремий модуль, який підключається до MCU. Наприклад, Wiznet, ENC28J60, чи вбудований контролер в деяких моделях мікроконтролерів.
- **Живлення та фізичний інтерфейс**. Необхідно забезпечити живлення для Ethernet-модуля/контролера, а також переконатися, що для наявне фізичне підключення до мережі за допомогою відповідного інтерфейсу (наприклад, RJ45).
- **Firmware для Ethernet-стеку**. Якщо MCU не має вбудованого Ethernet-стеку – потрібно його створити або використовувати сторонні бібліотеки.
- **Код для взаємодії з Ethernet.** Програма на MCU повинна мати відповідний код для ініціалізації Ethernet, встановлення параметрів мережі, обробки отриманих і надісланих даних.

> **Примітка.** Зазвичай в документації від виробників контролерів та відповідних Ethernet-модулів можна ознайомитись зі зразками коду та прикладами.

Приклад:

У середовищі Arduino комунікація по Ethernet зазвичай реалізується за допомогою бібліотеки Ethernet.h. Вона дозволяє підключатися до мережі, використовуючи, як DHCP, так і статичну конфігурацію IP-адреси. Оскільки Arduino спілкується з Ethernet-модулем за допомогою SPI – доцільно також використовувати бібліотеку SPI.h.

Ось кілька кроків для налаштування комунікації по Ethernet:

1) Підключити модуль Ethernet через відповідний інтерфейс. Інформацію про піни можна знайти у документації модуля.
2) Підключити бібліотеки для Ethernet та SPI:

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

3) Задати бажану MAC-адресу:

```
// MAC-адреса Ethernet модуля
byte mac[] = { 0xB8, 0x55, 0xEA, 0x73, 0x26, 0xC4 };
```

Можна використовувати випадкові адреси, але важливо, щоб вони були унікальними в локальній мережі.

4) Конфігурація IP-адреси. Для отримання або налаштування IP-адреси можна використовувати один з методів:
   - DHCP (автоматичне отримання IP від DHCP-сервера):

```
if (Ethernet.begin(mac) == 0)
{
  Serial.println("Не вдалося налаштувати Ethernet за допомогою DHCP ");
}
```

- Статична конфігурація:

```
byte ip[] = { 192, 168, 1, 100 };
Ethernet.begin(mac, ip);
```

Примітка. Ініціалізація модуля Ethernet виконується за допомогою **`begin`**. Детальніше про синтакс та параметри **`begin`** можна дізнатись за посиланням [8].

5) Написання програми для роботи з мережею. Вона може включати відправку даних на інші IP-адреси, надсилання запитів на сервер або отримання та обробку даних. Приклад програми для веб-клієнта можна переглянути за посиланням [9].

Wi-Fi

Wi-Fi (Wireless Fidelity) – це технологія бездротового обміну даними між пристроями за допомогою радіохвиль. В IoT, Wi-Fi дозволяє речам встановлювати бездротове з'єднання між собою та з точками доступу до мережі (AP).

Wi-Fi працює на частотах 2,4 ГГц та 5 ГГц і використовує різні стандарти, такі як 802.11b, 802.11g, 802.11n, 802.11ac, 802.11ax, які визначають швидкість передачі даних та інші параметри. Залежно від стандарту та середовища, Wi-Fi може працювати на різних відстанях, від кількох метрів до кількох сотень метрів.

Для захисту від несанкціонованого доступу та шифрування переданих даних можуть використовуватись різні протоколи шифрування, наприклад, WPA2 (Wi-Fi Protected Access 2) або WPA3.

Для підключиння вбудованої системи до мережі за допомогою Wi-Fi знадобиться наступне:

- **Wi-Fi модуль або SoC з Wi-Fi.** SoC (System on a Chip) включає в себе повний набір компонентів, таких як процесор, пам'ять, графічний процесор, контролери введення/виведення (I/O), модулі зв'язку (наприклад, Wi-Fi, Bluetooth), аналогові і цифрові блоки та інше.
- **Живлення та інтерфейс підключення**. Слід переконатися у правильному підключенні модуля та його антени, а також у наявності достатнього живлення для його роботи.
- **Драйвери та бібліотеки.** Для взаємодії з обраним модулем необхідно використовувати відповідний драйвер. Наприклад, для ESP8266 або ESP32, доволі популярною є бібліотека WiFi.h.
- **Код для взаємодії з Wi-Fi.** Написання коду для ефективної взаємодії з Wi-Fi включає реалізацію ініціалізації Wi-Fi, обробки подій та передачі даних.

Приклад:

Використання бібліотеки WiFi.h [10] робить процес створення проектів з бездротовим з'єднанням доволі простим. Підключення ESP32 до мережі Wi-Fi включає:

1) Завантаження і встановлення бібліотеки WiFi.h, якщо вона не була встановлена автоматично.
2) Її підключення в коді:

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

3) Встановлення інформації про назву мережі (SSID) та пароль до неї:

```
const char* ssid = "НАЗВА_МЕРЕЖІ";
const char* password = "ПАРОЛЬ_МЕРЕЖІ";
```

4) Вибір режиму роботи в якості станції (коли пристрій підключається до точки доступу Wi-Fi):

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

Також можливо працювати в інших режимах, а саме:

- режим точки доступу (коли пристрій стає точкою доступу Wi-Fi):

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

- змішаний режим (коли пристрій може одночасно підключатися до іншої точки доступу Wi-Fi та бути точкою доступу).

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

5) Виклик функції **begin** з аргументами передавши **ssid** та **password**, для початку підключення до мережі:

```
WiFi.begin(ssid, password);
```

6) Перевірити наявність підключення можливо за допомогою **WiFi.status()**, якщо з'єднання встановлено – результатом буде **WL_CONNECTED**:

```
while (WiFi.status() != WL_CONNECTED)
{
  //Код для роботи в мережі
}
```

7) Написання програми для роботи з мережею.

Примітка. Детальніше про можливості, що надає бібліотека WiFi.h можна дізнатись за посиланням [10].

Запитання для самоперевірки

1. Що таке Інтернет речей (IoT)?
2. Які особливості характеризують Інтернет речей?
3. Які тенденції у впровадженні IoT? Назвіть поширені галузі застосування.
4. Які типи мереж використовуються в IoT?
5. Опишіть найпростішу архітектуру IoT.
6. Опишіть екосистему IoT.
7. Через які області проходять зібрані з давачів дані?
8. Що таке гранична область (Edge)?
9. Які типи взаємодії існують в IoT?
10. Що потрібно для підключення вбудованих систем до мережі?

Тести

1. В яких типах мереж замість дротів використовуються електромагнітні хвилі?
   A) Ethernet.
   B) Бездротові мережі.
   C) Хмарні.
   D) Інтернет.
2. Збір даних про фізичні параметри, такі як температура, вологість, тиск виконується:
   A) Актуаторами.
   B) Розумними пристроями.
   C) Давачами.
   D) Протоколами.
3. Мільярди вбудованих систем і датчиків підключених до Інтернету це:
   A) Інтернет речей.
   B) Центр обробки даних .
   C) Edge-мережа.
   D) Хмара.
4. Яка технологія дозволяє бездротовим пристроям підключатися до інших пристроїв на невеликій відстані?
   A) Межа.
   B) Bluetooth.
   C) Ethernet.
   D) 4G/5G.
5. Який терміном описують мережу, що об’єднує інші мережі?
   A) Персональна мережа (PAN).
   B) Глобальна мережа (WAN).
   C) Локальна мережа (LAN).
   D) Інтернет.
   E) Інтранет.
6. Який тип бездротової мережі використовується для підключення пристроїв у межах міста для створення міської мережі (MAN)?
   A) ZigBee.
   B) Bluetooth.
   C) WiMAX.
   D) Wi-Fi.
7. Який тип мережі використовується для підключення компанії, розташованої в одному районі міста, до іншої локації в іншому місті?
   A) Локальна мережа (LAN).
   B) Глобальна мережа (WAN).
   C) Персональна мережа (PAN).
   D) Міська мережа (MAN).

8. Який тип мережі дозволяє персональному фітнесу-трекеру підключатися до смартфона?
   A) Персональна мережа (PAN).
   B) Глобальна мережа (WAN).
   C) Інтернет.
   D) Локальна мережа (LAN).
9. Який тип обчислень дозволяє опрацьовувати локальні дані на межі мережі?
   A) Хмарні.
   B) Граничні.
   C) Сенсорні.
   D) Бездротові.
10. Який тип обчислень знаходиться на периферії комерційної або корпоративної мережі, і чи це дає змогу попередньо обробляти дані, отримані від датчиків?
   A) Бездротові.
   B) Глобальні.
   C) Граничні.
   D) Інтернет.

ТЕМА 5. КОМУНІКАЦІЯ НА ПРИКЛАДНОМУ РІВНІ

План лекції

- Протоколи прикладного рівня в IoT
- Клієнт-серверна архітектура та Hypertext Transfer Protocol (HTTP)
- Серіалізація та десеріалізація даних
- Модель взаємодії публікація-підписка (Publish-Subscribe, Pub/Sub)
- Налаштування комунікації з використанням MQTT (Message Queuing Telemetry Transport)
- Якість обслуговування

Текстова частина лекції

Особливості протоколів прикладного рівня

На кожному рівні моделі OSI протоколи виконують свої унікальні функції та завдання. Протоколи рівня застосунків (Application Layer) визначають спосіб, яким застосунки взаємодіють між собою. Вони відповідають за такі високорівневі аспекти комунікації:

- **Стандартизація**. Визначення стандартів та правил обміну даними між різним пристроями, що дозволяють їм розуміти один одного та ефективно взаємодіяти.
- **Формат даних**. Визначення структури та формату даних, що передаються між пристроями, забезпечуючи узгодженість і сумісність.
- **Обробка запитів та відповідей**. Багато протоколів рівня застосунків базуються на концепції "запит-відповідь", що дозволяє клієнтським програмам запитувати дані або послуги від серверів, а серверам відповідати на ці запити відповідно до визначених протоколом правил.
- **Керування сеансом та комунікацією**. Деякі протоколи включають механізми для встановлення, управління та завершення сеансів комунікації.
- **Безпека та аутентифікація**. Багато протоколів включають механізми безпеки та аутентифікації, для забезпечення конфіденційності, цілісності та надання доступу до даних тільки для авторизованих пристроїв.

У сфері Інтернету речей існує багато різноманітних протоколів через специфічні вимоги та різноманітні умови, в яких працюють IoT-пристрої. Основними причинами цьому є:

- **Різноманітність пристроїв.** IoT-пристрої варіюються від простих датчиків до складних промислових машин, кожен з яких має свої вимоги до енергоефективності, потужності та обсягу пам'яті.
- **Різні сценарії використання.** Від домашньої автоматизації до промислових додатків.
- **Різні вимоги до даних.** Деякі додатки IoT потребують передачі невеликих обсягів даних, тоді як інші генерують великі дані і потребують частого оновлення.
- **Безпека та конфіденційність.** Наприклад, медичні пристрої чи системи спостереження, можуть вимагати високого рівня безпеки, що призводить до вибору протоколів з вбудованими механізмами шифрування та автентифікації.

- **Сумісність та стандартизація.** Протоколи повинні забезпечувати сумісність між пристроями від різних виробників та відповідати стандартам різних консорціумів. Також вони можуть бути оптимізовані для різних архітектур чи операційних систем.
- **Еволюція технологій.** IoT швидко розвивається і постійно з'являються нові технології, що призводить до появи нових протоколів.

Популярні протоколи в IoT

Нижче наведено деякі з найбільш поширених протоколів, які використовуються в IoT:

- **Hypertext Transfer Protocol (HTTP)** – один з найпопулярніших протоколів в Інтернеті. Використовується для обміну даними між веб-серверами та клієнтами. Працює за схемою "запит-відповідь", тобто кожен клієнтський запит отримує відповідь від сервера. Може споживати більше енергії за інші, більш оптимізовані, протоколи. Доцільно застосовувати в областях, де потребується передача великих обсягів даних.
- **Extensible Messaging and Presence Protocol (XMPP)** – гнучкий та розширюваний протокол. Однією з ключових особливостей є здатність до миттєвого обміну повідомленнями (напр. Google Hangouts) та визначення присутності учасників «розмови». XMPP присвоює пристроям ідентифікатори, схожі на адреси електронної пошти, що дозволяє їм однозначно ідентифікуватися та взаємодіяти один з одним. Часто використовується для T2T та T2S комунікації.
- **Message Queuing Telemetry Transport (MQTT)** – завдяки своїй ефективності, а відповідно – низькому енергоспоживанню, це один з найпопулярніших протоколів для передачі повідомлень в IoT. Цей «легкий» протокол працює поверх TCP/IP за моделлю видавець-підписник (pub/sub). Може працювати в умовах з частою втратою зв’язку.
- **Constrained Application Protocol (CoAP)** – призначений для пристроїв з обмеженими ресурсами. За допомогою нього такі пристрої можуть спілкуватися з рештою Інтернету. Він є бінарним аналогом протоколу HTTP.
- **SOAP (Simple Object Access Protocol)** в IoT використовується для обміну структурованими та стандартизованими повідомленнями (в форматі XML) між різнорідними пристроями, а також для віддаленого виклику процедур. Використовується у великих та складних системах IoT, де важлива структурованість та розширюваність повідомлень.
- **Data Distribution Service (DDS)** – використовується для розподіленого обміну даними в реальному часі. Працює за моделлю видавець-підписник, як і MQTT. Застосовується коли важлива швидкість, продуктивність та точність взаємодії, наприклад, для систем реального часу.
- **WebSocket** – дозволяє встановлювати двосторонній зв'язок між клієнтом та сервером, забезпечуючи повнодуплексну та негайну комунікацію в реальному часі. Використовується для взаємодії в інтерактивних IoT-застосунках, де важливий миттєвий обмін інформацією.

В мережі Інтернет можливо зустріти багато дискусій та досліджень, присвячених виявленню «найкращого» протоколу для конкретних задач.

Клієнт-серверна архітектура

Клієнт-серверна архітектура – це архітектурний шаблон програмного забезпечення для мережевої комунікації. Вона є однією із основних архітектур для організації взаємодії в IoT. У ній розрізняють два основних типи вузлів, клієнти та сервери:

- **Серверами (Server)** в можуть бути пристрої або хмарні сервіси, які забезпечують певні послуги чи функціонал. Вони готові слухати та відповідати на запити від клієнтів. Сервери можуть збирати, обробляти та зберігати дані, а також взаємодіяти з іншими пристроями в мережі.
- **Клієнти (Client)** – це пристрої або додатки, які використовують послуги або дані, надані серверами. Клієнти можуть відправляти запити до серверів для отримання інформації або виконання конкретних дій.

Процес взаємодії виглядає так:

1) Клієнти ініціюють взаємодію, надсилаючи запити на сервер.
2) Сервер обробляє ці запити і відправляє відповіді клієнтам.
3) Клієнти обробляють відповіді та виконують відповідні дії.

Ця модель ефективна для розподіленої архітектури, оскільки дозволяє доволі просто розділити певну функціональність або завдання між декількома пристроями.

HTTP

При використанні HTTP, у контексті IoT, існують певні особливості та обмеження:

- **Відомий та широко підтримуваний.** Є стандартним протоколом для веб-комунікацій, який підтримується великою кількістю пристроїв і платформ, тому більшість розробників добре знайомі з ним.
- **Робота на основі моделі запит-відповідь.** Це проста і зрозуміла модель, але вона може бути неефективною для деяких IoT-додатків, які вимагають постійного потоку даних або миттєвого зворотного зв'язку.
- **Складність та енерговитрати.** Кожне HTTP-повідомлення містить заголовки, які можуть бути значними за розміром. Використання поверх стеку TCP/IP вимагає встановлення та підтримання з'єднання, що додає накладні витрати і затримки, особливо у випадку нестабільних мереж.
- **Безпека**. Використання HTTPS (HTTP Secure) додає шифрування для захисту даних. Також підтримуються механізми аутентифікації і авторизації, проте їх реалізація може бути складною через обмежені ресурси IoT-пристроїв та специфічні умови їх роботи.

HTTP має кілька основних версій, які еволюціонували з часом:

- HTTP/0.9 (1991 рік) та HTTP/1 (1996 рік) – TCP з’єднання створювалось для кожного окремого запиту та відповіді.
- HTTP/1.1 (1997 рік) – додано можливість використовувати одне з'єднання для відправлення кількох запитів (keep-alive), а також введено поняття "трубопроводу" (pipelining) для послідовної обробки запитів.

- HTTP/2 (2015 рік) – значні поліпшення швидкості та ефективності передачі даних. Введено концепцію багатопоточності, що дозволяє одному з'єднанню обробляти кілька одночасних запитів.
- HTTP/3 (2022 рік) ще більше підвищив швидкість і ефективність, транспортний протокол TCP було замінено на QUIC (на основі на UDP). В деяких випадках HTTP/3 втричі швидший за HTTP/1.1 (більшість веб-сайтів досі працюють лише з HTTP/1.1).

Методи в HTTP – це спеціальні команди, які визначають, яку дію клієнт хоче виконати з визначеним ресурсом на сервері, наприклад:

- **GET** – отримання ресурсів з сервера.
- **POST** – надсилання даних на сервер.
- **PUT** – оновлення чи створення ресурсу.
- **DELETE** – видалення ресурсу.

Про результат обробки запиту клієнт повідомляється за допомогою спеціальних кодів.

Коди стану HTTP – це тризначні числові відповіді, що веб-сервери повертають клієнтам для інформування про результат обробки запиту. Вони поділяються на кілька категорій:

- **1xx** (інформаційні) – інформують, що сервер отримав початкову частину запиту та клієнт може продовжувати відправляти залишок даних.
- **2xx** – запит успішно отримано, зрозуміло і прийнято (наприклад, 200 OK).
- **3xx** (перенаправлення) – потрібні додаткові дії для завершення запиту (наприклад, 301 Moved Permanently).
- **4xx** – помилка на стороні клієнта (наприклад, 404 Not Found).
- **5xx** – помилка на стороні сервера (наприклад, 500 Internal Server Error).

Комунікація по HTTP

Допустимо, є система для вимірювання температури та вологості в приміщенні. Використаємо HTTP-методи для взаємодії з сервером:

- Пристрій вичначає температуру та вологість і надсилає дані серверу за допомогою методу POST для збереження чи оновлення значень. Наприклад:

```
POST /temperature.php HTTP/1.1
Host: example.com
Content-Type: application/json; charset=utf-8
Content-Length: 36

{
  "temperature": 25.5,
  "humidity": 50.2
}
```

- У цього ж сервера, користувач може запитати про поточні значення температури та вологості використовуючи метод GET, наприклад:

```
GET /temperature.php HTTP/1.1
Host: example.com
```

- Якщо деякий ресурс (напр. temperature.php) не є актуальним чи більше не потрібен, клієнт може використовувати DELETE для його видалення:

```
DELETE /temperature.php HTTP/1.1
Host: example.com
```

Заголовок "Host" вказує сервер, до якого адресований запит (може бути у вигляді IP-адреси), а "Content-Length" вказує на довжину тіла запиту в байтах. У реальних сценаріях включення різних заголовків є важливим для коректної обробки запиту сервером.

Мінімалістична відповідь від сервера, в вигляді простого тексту, може мати наступний вигляд:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8

Hello! Current temperature = 25, humidity = 50!
```

Серіалізація даних

Серіалізація – це процес перетворення об'єктів або структур даних в формат, який можна зберегти або передати. Наприклад, є об’єкт, який має наступні поля:

```
int deviceId = 10;
double temperature = 25.5;
double humidity = 50;
String city = "Kyiv";
double coordinates[] = {50.450001, 30.523333};
```

Для передачі цієї інформації по мережі, можна перетворити її в рядок тексту:

```
String result = String(deviceId) + String(temperature, 1) + String(humidity) + city
    + String(coordinates[0], 6) + String(coordinates[1], 6);
```

Результатом такої «серіалізації» буде:

```
1025.550Kyiv50.45000130.523333
```

Це вже можливо передати, але отримувачу буде складно розібратися у тексті, який представлений у вигляді одного довгого рядка без чіткого розділення між різними значеннями. Визначити, яке значення відповідає ідентифікатору пристрою, температурі, вологості, місту та координатам можна лише на основі припущень, наперед знаючи про структуру даних, про що часто може бути невідомо отримувачу.

Для кращої серіалізації існує багато популярних форматів, наприклад:

- **XML** (Extensible Markup Language) – це розширювана мова розмітки, що використовує теги для визначення структури даних. В такому форматі, представлений раніше текст, може виглядати наступним чином:

```
<sensorData>
  <deviceId>10</deviceId>
  <temperature>25.5</temperature>
  <humidity>50</humidity>
  <city>Kyiv</city>
  <coordinates>
```

```
    <latitude>50.450001</latitude>
    <longitude>30.523333</longitude>
  </coordinates>
</sensorData>
```

В ньому кожне значення огорнуто в спеціальні теги, проаналізувавши які, його можна доволі легко віднайти і зрозуміти.

- **JSON** (JavaScript Object Notation) є одним з найпоширеніших форматів для серіалізації даних. Цей текстовий формат зручний для читання як людиною, так і комп'ютером. Він дуже часто застосовується для передачі даних між веб-клієнтами та серверами.

```
{
    "deviceId": 10,
    "temperature": 25.5,
    "humidity": 50,
    "city": "Kyiv",
    "coordinates": [50.450001, 30.523333]
}
```

JSON має більш простий синтаксис, порівняно з XML, та є більш економічним у використанні ресурсів. Текст складається з ключів та значеннь, що поєднані в пари за допомогою двокрапок, де кожна пара відокремлюється комою, а весь об'єкт обмежується фігурними дужками.

Масиви в JSON дозволяють представляти упорядковані колекції даних та структурувати і організовувати інформацію зручним способом. Вони відображаються у вигляді послідовностей значень, розділених комами, та укладених у квадратні дужки. Кожен елемент масиву може бути будь-якого типу даних: числом, рядком, об'єктом, масивом, булевим значенням або null.

Десеріалізація

Десеріалізація – це процес перетворення представлення даних зі збереженого формату (наприклад, рядка JSON або XML) у структуру даних, зрозумілу для програми чи системи. Цей процес використовується для відновлення серіалізованих даних і їх подальшого використання у програмі. Наприклад, при отриманні через мережу рядка тексту в форматі JSON, десеріалізація перетворить його на певну структуру даних (набір окремих змінних, масивів і т.д.), з якими легко працювати в програмі.

Для десеріалізації можна застосовувати декілька підходів:

- Використання вбудованих функцій (**`indexOf`** та **`substring`**) для роботи з рядками, що використовуються для пошуку та виділення певної частини символів з рядка. Цей підхід може бути ефективним для десеріалізації простих текстових даних, але стає складним і неефективним для більш складних структур даних.
    - **`indexOf(`**`char` ***`ch`***`)` знаходить індекс заданого символу ***`ch`*** в рядку або повертає -1, якщо такого символу не знайшлося.
    - **`substring`**`(unsigned int` ***`from`***`, unsigned int` ***`to`***`)` витягує підрядок від початкового індексу ***`from`*** (номеру символа в базовому реченні) до кінцевого індексу ***`to`*** (не включаючи символ на позиції ***`to`***).

Приклад затосування `indexOf` та `substring`:

```
// Дані отримані з мережі
String data =
"{\"deviceId\":44,\"temperature\":25.5,\"humidity\":50,\"city\":\"Kyiv\",\"coo
rdinates\":[50.450001,30.523333]}";

// Знаходимо початковий індекс значення поля "deviceId" (2)
int deviceIdStart = data.indexOf("deviceId") + 10;
// До значення 2 додаємо довжину deviceId": - 10, що включає 8 символів,
двокрапку і пробіл

// Визначаємо індекс коми, що йде після значення "deviceId" (14)
int deviceIdEnd = data.indexOf(",", deviceIdStart);

// Витягуємо підрядок від дванадцятого до чотирнадцятого символу
int deviceId = data.substring(deviceIdStart, deviceIdEnd).toInt();

// В результаті змінна deviceId буде дорівнювати 44
```

Примітка. Символ **** (екрануючий символ) використовується для включення спеціальних символів, таких як лапки (**"**), у рядок. Коли компілятор бачить **\"**, він інтерпретує це як один символ **"** у рядку, а не два окремих символи **** та **"**. Тому функція **`indexOf`** не враховує екрануючі символи.

- Використання бібліотек для роботи з різними форматами даних. На відміну від застосування лише функцій **`indexOf`** та **`substring`**, бібліотечні варіанти зазвичай надійніші, ефективніші і зручніші. Для роботи з JSON, однією з таких бібліотек є ArduinoJson [11].

Приклад затосування ArduinoJson:

```
#include <ArduinoJson.h> // Підключення бібліотеки

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

  // Серіалізовані дані у форматі JSON
  String json_data = "{\"deviceId\":44, \"temperature\":25.5, \"humidity\":50,
\"city\":\"Kyiv\", \"coordinates\":[50.450001, 30.523333]}";

  // Створення об'єкта для зберігання десеріалізованих даних
  JsonDocument doc;

  // Десеріалізація JSON-рядка в об'єкт JsonDocument
  deserializeJson(doc, json_data);

  // Отримання десеріалізованих значень з JSON-об'єкта
  int deviceId = doc["deviceId"];
  double temperature = doc["temperature"];
```

```
  double humidity = doc["humidity"];
  const char* city = doc["city"];
  double lat = doc["coordinates"][0];
  double lon = doc["coordinates"][1];

  // Виведення отриманих значень на серійний монітор
  Serial.print("Ідентифікатор пристрою: ");
  Serial.println(deviceId);
  Serial.print("Температура: ");
  Serial.println(temperature);
  Serial.print("Вологість: ");
  Serial.println(humidity);
  Serial.print("Місто: ");
  Serial.println(city);
  Serial.print("Широта: ");
  Serial.println(lat, 6); // Виводимо координати з 6 знаками після коми
  Serial.print("Довгота: ");
  Serial.println(lon, 6); // Виводимо координати з 6 знаками після коми
}

void loop()
{
  // Основний цикл залишається порожнім, оскільки все виконання відбувається в
setup()
}
```

В результаті обробки десеріалізованих даних в серійному моніторі буде відображено наступне:

```
Ідентифікатор пристрою: 44
Температура: 25.50
Вологість: 50.00
Місто: Kyiv
Широта: 50.450001
Довгота: 30.523333
```

Приклад програми для обміну даними по HTTP

Для виконання запиту до сервера в мережі Інтернет та обробки відповіді від нього, за допомогою ESP32, необхідно виконати такі кроки:

1) Підключення бібліотек. WiFi.h – для роботи в Wi-Fi мережі, та HTTPClient – для роботи за протоколом HTTP:

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

2) Визначення назви та паролю Wi-Fi мережі за допомогою відповідних констант **ssid** та **password**:

```
const char* ssid = "Моя_точка_доступу";
const char* password = "Мій_пароль";
```

3) Визначення адреси сервера для подальшого виконання GET-запиту до нього:

```
const String server = "http://example.com";
```

4) Встановлення підключення до Wi-Fi мережі в функції **setup** з використанням циклу **while** для очікування успішного підключення:

```
void setup()
{
  // Ініціалізація порту для виведення даних у консоль
  Serial.begin(115200);

  // Підключення до Wi-Fi
  WiFi.begin(ssid, password);

  while(WiFi.status() != WL_CONNECTED)
  {
    delay(1000);
    Serial.println("Підключення до Wi-Fi...");
  }

  Serial.println("Підключення успішне");
}
```

5) На початку кожної ітерації **loop** виконується перевірка наявності підключення до Wi-Fi:

```
void loop()
{
    //Перевірка наявності Wi-Fi підключення
    if(WiFi.status()== WL_CONNECTED)
    {
      //Код для роботи по HTTP, описаний в пунктах 6-9
    }
    else
    {
      Serial.println("З'єднання Wi-Fi втрачено");
    }
}
```

6) Якщо з'єднання Wi-Fi встановлено, створюється об'єкт класу **HTTPClient** та виконується його налаштування. В якості аргументу методу **begin** встановлюється адреса, до якої буде виконуватись запит. Методу **http.GET()** використовується для виконання GET-запиту. Цей метод повертає код відповіді типу **int**, для подальшої роботи з яким, він поміщається в відповідну змінну:

```
HTTPClient http;

http.begin(server);

int responseCode = http.GET();
```

7) Перевірка, чи отримано позитивний HTTP-код відповіді (наприклад, 200). Якщо так, виклик блок коду всередині **if**. В консоль виводиться значення коду відповіді. Далі, викликається метод **getString()**, який повертає вміст відповіді у вигляді рядка **(String)**. Цей вміст зберігається у **payload**, після чого також виводиться у консоль:

```
if (responseCode > 0)
{
  Serial.print("HTTP код відповіді: ");
  Serial.println(responseCode);
  String payload = http.getString();
  Serial.println(payload);
}
```

8) Якщо код відповіді менше або дорівнює 0, викликається блок коду всередині **else**, який повідомляє про помилку:

```
else
{
  Serial.print("Error code: ");
  Serial.println(responseCode);
}
```

9) Завершення роботи з **HTTPClient** та затримка перед наступним запитом:

```
http.end();
//30 хв очікування до наступного запиту
delay(1800000);
```

Цей код дозволяє виконувати періодичні GET-запити до вказаного сервера через Wi-Fi та виводити отриману від нього інформацію у консоль.

У зв'язку із зростанням розміру та складності IoT-систем, іноді HTTP може бути «важким», для певних застосувань, через свою залежність від текстового представлення даних та наявність значних накладних витрат. Тому, доцільно розглянути альтернативи, наприклад, більш легкий та ефективний – MQTT.

Протокол MQTT

MQTT (Message Queuing Telemetry Transport) – протокол комунікації, призначений для передачі повідомлень між пристроями у мережах Інтернету речей. Він використовує менше ресурсів порівняно з HTTP, що дозволяє йому ефективніше працювати на малопотужних пристроях.

В порівнянні з HTTP, MQTT використовує менші за обсягом повідомлення з меншою кількістю заголовків, що призводить до зменшення трафіку мережі. Крім цього, при взаємодії по HTTP сервер може відправляти дані лише у відповідь на запит клієнта (принцип "request-response"), що не сприяє ініціативному відправленню даних від сервера до клієнта без попереднього запиту. Це стає проблемою, зокрема, в сценаріях дистанційного керування в IoT, де може бути потрібно активно взаємодіяти з пристроями, відправляючи їм вказівки або оновлення. У такому випадку, MQTT, який підтримує "push", тобто, може ініціювати відправлення даних від сервера до клієнта, буде більш ефективним і практичним вибором.

Модель взаємодії Publish-Subscribe (Pub/Sub, Видавець-Підписник)

MQTT працює на основі моделі взаємодії Publish-Subscribe. Ця модель дозволяє розподіленим клієнтам обмінюватися повідомленнями через брокера (broker), який відповідає за керування таким обміном.

Основні поняття та учасники MQTT-моделі:

- **Брокер (Broker)** – центральний елемент системи, що приймає і маршрутизує повідомлення між клієнтами. Брокер відповідає за обробку публікацій і підписок, а також управлінням якістю обслуговування (QoS).
- **Видавець (Publisher)** – клієнт, який публікує (відправляє) повідомлення на конкретний топік (тему) до брокера. Видавцем може бути будь-який пристрій або програма, яка може генерувати інформацію.
- **Підписник (Subscriber)** – клієнт, який підписується на отримання повідомлень з певного топіка. Після підписки він буде отримувати всі повідомлення, які публікуються в цей топік.
- **Топік (Topic)** – це іменований канал, в який можна щось опублікувати або підписатися на нього. Топіки дозволяють класифікувати інформацію і визначають, хто може її отримувати.
- **Повідомлення (Message)** – це одиниця інформації, яку видавець публікує на брокері, і яку можуть отримувати всі підписники.

Робочий процес в моделі публікації-підписки наступний:

1) **Публікація (Publishing)** – видавець (Publisher) публікує повідомлення у визначений топік брокера.
2) **Підписка (Subscribing)** – підписник (Subscriber) повідомляє брокера про перелік топіків з яких він бажає отримувати всі повідомлення.
3) **Маршрутизація повідомлень (Message routing)** – брокер отримує повідомлення від видавця і маршрутизує його до всіх підписників, які підписалися на відповідний топік.
4) **Отримання повідомлень (Receiving)** – підписники отримують повідомлення від брокера, перевіряють з яких топіків вони надійшли та виконують подальшу обробку даних.

Модель публікації-підписки уніфікує обмін даними в розподіленій системі, забезпечуючи ефективну і асинхронну комунікацію між мережевими пристроями.

Налаштування комунікації з використанням MQTT

Протокол MQTT надає певні можливості для безпечної передачі даних, які включають: TLS/SSL шифрування; аутентифікацію; авторизацію; безпечні топіки; повідомлення (Last Will) в разі аварійного відключення та інше.

Налаштування комунікації з використанням MQTT складається з наступних кроків:

1) **Вибір та налаштування MQTT-брокера.** Є багато різних реалізацій брокерів, такі як: Mosquitto, RabbitMQ, HiveMQ, EMQX та інші. Оберіть та налаштуйте брокер для ваших потреб.

2) **Налаштування MQTT-клієнтів (видавця та підписників).** Визначте, які пристрої або програми будуть виступати у ролі MQTT-клієнтів. Це можуть бути додатки на мобільних пристроях, мікроконтролери, сервери або інше обладнання. Виберіть відповідну бібліотеку чи фреймворк для відповідної платформи та мови програмування.
3) **Підключення до брокера.** Використовуючи функції бібліотеки вкажіть адресу та порт брокера, а також інші параметри, за потреби, та встановіть підключення.
4) **Публікація повідомлень (Publish).** Сформуйте повідомлення та передайте його в якості аргументу у функцію публікації, яка зазвичай має два параметра – текст повідомлення та назва топіку, в який воно буде опубліковане. Не слід публікувати повідомлення занадто часто, це може спричинити відмову в обслуговуванні.
5) **Підписка на топіки (Subscribe).** Функція підписки зазвичай має лише один парамер – назву топіка, на який необхідно підписатись. Але клієнт може підписуватись на безліч топіків одночасно, інколи, навіть на різних брокерах.
6) **Обробка отриманих повідомлень.** Проаналізувавши зміст отриманого повідомлення та назву топіка, з якого воно надійшло, можна по різному на це реагувати, наприклад: змінювати стан пристрою; відображати інформацію в інтерфейсі користувача або виконувати будь-які інші дії.

Приклади програм для Publisher та Subscriber пристроїв

Розглянемо обмін повідомленнями між двома платами – одна буде видавцем (відправником повідомлень), а інша підписником (отримувачем повідомлень). Для роботи за протоколом MQTT використовуватиметься бібліотека PubSubClient. Детальніше про неї можна дізнатись з документації за посиланням [12].

В якості брокера використовуватиметься безкоштовний публічний MQTT брокер від HiveMQ (детальніше за посиланням [13]). Налаштування якого виглядають наступним чином:

- Server: broker.hivemq.com
- TCP Port: 1883 (Websocket Port: 8000; TLS TCP Port: 8883; TLS Websocket Port: 8884)

Перевірити зміст топіків, та протестувати публікацію повідомлень можливо з браузера, за допомогою онлайн-клієнту від HiveMQ [14].

Основні кроки, необхідні для налаштування видавця (Publisher):

1) Підключення необхідних бібліотек: WiFi.h для роботи з Wi-Fi та PubSubClient.h для роботи з MQTT.

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

2) Налаштування параметрів мережі та MQTT. Задаються параметри для підключення до Wi-Fi, такі як SSID та пароль, а також параметри для MQTT, такі як адреса брокера, порт, топік та ідентифікатор клієнта (для приватності додатково можливо використовувати логін та пароль). Кожен клієнт повинен мати унікальний **`clientId`**, це дозволяє брокеру відрізняти одного клієнта від іншого та визначати на які топіки вони підписані:

```
const char* ssid = "Wokwi-GUEST";
const char* password = "";
const char* mqttServer = "broker.hivemq.com";
```

```
const char* mytopic = "MyTopic/number1";
int port = 1883;
const char* clientId = "88";
```

3) Ініціалізація об'єктів типу WiFiClient та PubSubClient, які використовуються для взаємодії з Wi-Fi та MQTT відповідно:

```
WiFiClient wifiClient;
PubSubClient client(wifiClient);
```

4) З'єднання з Wi-Fi та MQTT. У функції **setup()** виконується підключення до Wi-Fi, а також, встановлюється адреса та порт MQTT-брокера для клієнта PubSubClient:

```
void setup()
{
  // Ініціалізація порту для виведення даних у консоль
  Serial.begin(115200);
  // Підключення до Wi-Fi
  WiFi.begin(ssid, password);
  Serial.print("Підключення до Wi-Fi...");
  while(WiFi.status() != WL_CONNECTED)
  {
    delay(300);
    Serial.print(".");
  }
  Serial.println();
  Serial.println("Підключення успішне");

  client.setServer(mqttServer, port);
}
```

5) У основному циклі програми (loop) перевіряється, чи встановлено з'єднання з MQTT-брокером. Якщо ні (або воно зникло), викликається функція **reconnect()**, що спробує підключитися знову. Якщо з’єднання є – у консоль виводиться інформація щодо топіку та тексту, який буде публікуватись в нього.

```
void loop()
{
    delay(1000);
    if (!client.connected())
    {
      reconnect();
    }
    Serial.print("Відправка повідомлення в топік ");
    Serial.print(mytopic);
    Serial.println(": ");
    Serial.println("Щось корисне");

    client.publish(mytopic,"Щось корисне");
}
```

6) Функція **reconnect()**, за потреби, перепідключається до MQTT-брокера. При невдалій спробі – у консоль виводиться повідомлення про помилку та виконується очікування перед повторним перепідключенням:

```
void reconnect()
{
  while (!client.connected())
  {
    Serial.println("Підключення по MQTT...");
    if (client.connect(clientId))
    {
      client.subscribe(mytopic);
      Serial.print("Підключений клієнт: ");
      Serial.println(clientId);
    }
    else
    {
      Serial.print("повторна спроба через 3 секунди. Помилка: ");
      Serial.println(client.state());
      delay(3000);
    }
  }
}
```

Основні кроки, необхідні для налаштування підписника (Subscriber):

1) Спочатку, як і у попередньому прикладі коду, необхідно: підключити бібліотеки, задати параметри мережі і MQTT та ініціалізувати відповідні об'єкти. Один пристрій (клієнт) може одночасно бути і видавцем і підписником, але, для іншого пристрою – використовується новий Id клієнта:

```
const char* clientId = "44";
```

2) З'єднання з Wi-Fi та MQTT. До попереднього варіанту функції **setup()** необхідно додати виклик **client.setCallback(callback)**, що вказує клієнту PubSubClient використовувати функцію **callback** для обробки повідомлень:

```
client.setServer(mqttServer, port);
client.setCallback(callback);
```

3) Підписник в функції **loop()** перевіряє наявність підключення до MQTT-брокера, а при отриманні нових повідомлень викликається функція callback. Невелику затримку (у цьому випадку, 30 мілісекунд) встановлено для зменшення навантаження на процесор та регулювання швидкості виконання циклу:

```
void loop()
{
  delay(30);
  if (!client.connected())
      reconnect();
  client.loop();
}
```

4) Функція **`callback:`** виводить у консоль інформацію про отримане повідомлення, вказуючи топік, з якого воно прийшло; зчитує байти повідомлення та конвертує їх в рядок (**`String`**); перевіряє, чи топік повідомлення співпадає з вказаним топіком (**`mytopic`**) і, якщо умова виконується, виводить отримане повідомлення у консоль:

```
void callback(char* topic, byte* message, unsigned int length)
{
  Serial.print("Отримано повідомлення з топіка ");
  Serial.print(topic);
  Serial.println(": ");
  String stMessage;

  for (int i = 0; i < length; i++)
  {
    stMessage += (char)message[i];
  }

  Serial.println(stMessage);

  if (String(topic) == mytopic)
  {
    Serial.println("Виконання чогось корисного");
  }
}
```

Наведений приклад реалізує мережеву взаємодію клієнта-підписника та клієнта-видавця з використанням публічного брокера. Публічні брокери зручні для швидкого початку роботи та тестування, але вони не гарантують конфіденційності та безпеки даних. Приватні брокери надають більше контролю, але вимагають більше зусиль для розгортання та управління.

Якість обслуговування MQTT

Слід зазначити, що протокол MQTT надає три рівні обслуговування (Quality of Service, QoS) для доставки повідомлень:

- **QoS 0** (максимум один раз) – повідомлення доставляється один раз, але без підтвердження та повторної передачі. TCP гарантує доставку до конкретної машини, але не до застосунку (MQTT-клієнта чи брокера).
- **QoS 1** (принаймні, один раз) – гарантується, що повідомлення буде доставлено принаймні один раз, але можливе отримання дублікатів, тобто може бути отримано декілька копій повідомлення.
- **QoS 2** (точно один раз) – гарантується доставка повідомлення лише один раз. Така точність доставки може вимагати більше ресурсів мережі.

Рівень QoS визначається відправником повідомлення. Для сценаріїв, де важлива точність доставки (наприклад, фінансові системи), рекомендується використовувати QoS 2. У випадку, коли допускається деяка втрата або дублювання повідомлень, можна використовувати менш вимогливі рівні QoS 0 або QoS 1.

Запитання для самоперевірки

1. Що таке протоколи прикладного рівня в IoT і чому вони важливі для взаємодії між пристроями?
2. Розкрийте концепцію рівневої структури IoT. Які моделі існують і на які рівні вони поділяються?
3. Як працює протокол HTTP? Які основні характеристики цього протоколу?
4. Що таке гіпертекст і HTML?
5. Навіщо потрібна серіалізація та десеріалізація даних? Які популярні формати тесктових даних ви знаєте?
6. Що таке протокол MQTT? Які його особливості та переваги в контексті Інтернету речей?
7. Які основні поняття відносяться до моделі обміну повідомленнями Видавець-Підписник?
8. Опишіть робочий процес в моделі Publisher/Subscriber.
9. В чому основні відмінності MQTT та HTTP?
10. Що таке якість обслуговування та які рівні QoS існують в MQTT?

Тести

1. Що таке QoS (англ. Quality of service, укр. Якість обслуговування) в контексті MQTT?
   A) Гарантія доставки повідомленнь.
   B) Доступності до мережі Інтернет.
   C) Типи підключень до MQTT брокера.
   D) Швидкості обміну даними за протоколом MQTT.
2. Які рівні якості обслуговування існують в протоколі MQTT?
   A) Basic, Intermediate, Advanced, Superior.
   B) Level 1, Level 2, Level 3, Level 4.
   C) QoS 0, QoS 1, QoS 2, QoS 3.
   D) QoS 0, QoS 1, QoS 2.
3. Яка з наступних характеристик відноситься до протоколу HTTP?
   A) Запит-відповідь, текстовий протокол.
   B) Маршрутизація мережевого трафіку.
   C) Управління станом.
   D) Аутентифікація користувачів.
4. Які методи використовуються в протоколі HTTP для виконання запитів до веб-ресурсів?
   A) GET, PUT, DELETE, POST.
   B) GET, UPDATE, RETRIEVE, SUBMIT.
   C) READ, WRITE, GET, EXECUTE.
   D) HTML, TCP, IP, SQL.
5. Які складові включає в себе відповідь HTTP?
   A) IP-адреса сервера, URL ресурсу, метод запиту.
   B) Метод запиту, тіло відповіді, заголовки.
   C) Код стану HTTP, заголовки відповіді, тіло відповіді.
   D) Тіло та заголовки відповіді, IP-адреса клієнта.

6. Які складові включає запит HTTP?
   A) IP-адреса сервера, URL ресурсу, код стану HTTP.
   B) Метод запиту, тіло відповіді, заголовки запиту.
   C) Метод запиту, URL ресурсу, заголовки запиту.
   D) Тіло запиту, заголовки, код статусу HTTP.
7. Які статуси може повертати сервер у відповідь на запити HTTP?
   A) 100, 200, 300, 400.
   B) 200, 300, 400, 500.
   C) 100, 200, 300, 500.
   D) 100, 200, 300, 400, 500.
8. Що означає термін "Message Queuing Telemetry Transport" (MQTT)?
   A) Протокол передачі повідомлень між різними серверами.
   B) Метод передачі повідомлень у веб-браузері.
   C) Протокол передачі повідомлень між пристроями.
   D) Система організації черг повідомлень на сервері.
9. Що представляє собою брокер у моделі Pub/Sub (Publisher/Subscriber)?
   A) Сервер для збереження бази даних.
   B) Клієнт, який одержує повідомлення від інших клієнтів.
   C) Програма, що приймає і маршрутизує повідомлення між клієнтами.
   D) Сервер, який підписується на отримання повідомлень з певного топіка.
10. Які особливості властиві моделі Pub/Sub (Publisher/Subscriber)?
    A) Кожен підписник має власного видавця.
    B) Підписники підписуються на окремі теми, а видавці надсилають повідомлення до цих тем.
    C) Підписники надсилають повідомлення серверу, а сервер розсилає їх іншим підписникам.
    D) Кожен клієнт не може бути і підписником, і видавцем.
11. З яких рівнів складається трирівнева архітектура IoT?
    A) Застосунків, мережевий, інтернет.
    B) Мережевий, маршрутизації, безпеки.
    C) Зв'язку, інтернет, застосунків.
    D) Сприйняття, мережевий, застосунків.
12. Яка архітектура полягає в тому, що багато пристроїв можуть звертатися до одного, "прислужуючого" пристрою, для отримання або надсилання даних?
    A) Потокова архітектура.
    B) Командна архітектура.
    C) Клієнт-серверна архітектура.
    D) Централізована архітектура.
13. Які протоколи належать до прикладного рівня?
    A) TCP.
    B) IP.
    C) HTTP.
    D) Wi-Fi.

14. Що означає термін "серіалізація даних"?
    A) Відправлення даних через послідовний порт.
    B) Кодування даних у формат, придатний для передачі або збереження.
    C) Зберігання даних у локальній базі даних.
    D) Шифрування даних перед передачею через мережу.
15. Що таке "JSON"?
    A) Формат для передачі та збереження структурованих даних.
    B) Алгоритм для стиснення зображень.
    C) Технологія для керування базами даних.
    D) Мова програмування для створення веб-сайтів.

ТЕМА 6. ВЕЛИКІ ДАНІ ТА ХМАРНІ ОБЧИСЛЕННЯ

План лекції

- Великі дані (Big data)
- Розподілена обробка даних
- Класифікація, функції та постачальники хмарних послуг
- Написання програми для роботи з хмарною платформою

Текстова частина лекції

Великі дані

Великі дані (Big Data) – це дані таких обсягів, які занадто великі, швидкі або складні для традиційних методів обробки. Вони включають як структуровані, так і неструктуровані дані, що надходять з різних джерел, таких як соціальні медіа, сенсори IoT, транзакції в інтернет-магазинах, тощо. Завдяки аналізу та обробці великих даних різні організації можуть отримувати цінні інсайти та приймати обґрунтовані рішення. Великі дані включають як структуровані так і неструктуровані дані.

Структуровані дані – мають чітко визначену організацію та формат. Зазвичай вони зберігаються у таблицях баз даних або інших структурованих форматах, таких як CSV або XML. Такі дані легко обробляти за допомогою реляційних баз даних та використовувати для аналізу і звітності.

Неструктуровані дані – не мають чіткої організації та можуть бути у будь-якому форматі. Це можуть бути текстові файли, електронні листи, зображення, відео, аудіофайли та інші неконвенційні формати. Їх складно обробляти та аналізувати за допомогою традиційних методів, але вони можуть містити цінну інформацію, що може бути використана за допомогою розширених аналітичних методів, таких як машинне навчання або обробка природної мови.

Можна виділити такі характеристики великих даних:

- **Обсяг** (Volume). Зберігаються та обробляються величезні обсяги даних, терабайти та петабайти даних.
- **Швидкість** (Velocity) зростяння обсягу даних постійно зростає.
- **Різноманітність** (Variety). Різними джерелами даних генеруються дані різних типів, включаючи текст, відео, аудіо, зображення, сенсорні дані та інше.

Скільки даних збирають датчики?

В сучасних вбудованих системах датчики здатні збирати величезну кількість даних, наприклад:

- Сенсори в промисловості можуть генерувати від кількох гігабайтів до терабайтів даних на день або навіть на годину.
- Смарт-сенсори в будинках (термостати, камери, системи безпеки) генерують менші об'єми, але все ж можуть створювати кілька гігабайтів даних на місяць.
- Автомобільні сенсори (в самокерованих автомобілях) можуть генерувати декілька гігабайтів даних на день завдяки використанню камер, лідара, радарів та інших сенсорів.

- Датчики двигунів в авіації можуть генерувати по 10 гігабайтів щосекунди та по петабайту за один політ.

Збір та обробка таких великих обсягів даних потребує потужних інструментів для їх передачі, аналізу та зберігання.

Розподілена обробка даних

З великими даними зазвичай працюють за допомогою мережі серверів (дата центрів), які географічно розташовані в різних місцях, але працюють разом, тобто відбувається розподілена обробка даних. Вона підвищує надійність, масштабованість та доступність даних і послуг, створюється ілюзія єдиного ресурсу – хмари.

Хмарні обчислення (англ. Cloud Computing) – це концепція, яка описує спосіб використання різноманітних обчислювальних ресурсів через Інтернет (таких як обчислювальна потужність, простір для зберігання даних, мережеві сервіси). Замість того, щоб встановлювати власні великі обчислювальні системи на місцевих серверах або комп'ютерах, користувачі можуть отримати доступ до них через мережу Інтернет (хмару).

Переваги використання Cloud Computing:

- **Масштабованість (Scalability).** Користувачі можуть динамічно коригувати виділення обчислювальних ресурсів. Наприклад, збільшувати потужність під час великої навантаженості та зменшувати під час менш інтенсивних періодів.
- **Доступність (Availability).** Сервіси хмарного обчислення зазвичай надають високий рівень доступності, оскільки вони розподілені між різними фізичними серверами та дата-центрами.
- **Гнучкість (Flexibility).** Користувачі можуть вибирати та конфігурувати обчислювальні ресурси, які вони використовують, відповідно до своїх потреб.
- **Економія витрат (Cost efficiency).** Хмарні обчислення дозволяють користувачам платити лише за ті ресурси, які вони використовують.
- **Спільний доступ (Shared access).** Ресурси можуть використовуватись кількома користувачами одночасно, але з дотриманням відповідних заходів безпеки та ізоляції.

Класифікація хмар

Хмари можна **класифікувати** за різними критеріями на основі видів послуг та структури розташування ресурсів. Ось декілька основних видів хмар:

1) **За видами послуг** (Service Models):
    - Інфраструктура як послуга, **IaaS** (Infrastructure as a Service) – надаються обчислювальні ресурси, такі як віртуальні машини, мережеві ресурси та сховища даних.
    - Платформа як послуга, **PaaS** (Platform as a Service) – надається платформа для розробки та розгортання програмного забезпечення без необхідності управління інфраструктурою.
    - Програмне забезпечення як послуга, **SaaS** (Software as a Service) – надаються готові застосунки та послуги доступні користувачам через Інтернет.

2) **За рівнем доступу** (Deployment Models):
   - Публічна хмара (Public cloud) – інфраструктура обслуговується та надається виробником хмарних послуг для різноманітних організацій одночасно.
   - Приватна хмара (Private cloud) – зазвичай, належить та використовується однією організацією або власником.
   - Гібридна хмара (Hybrid cloud) – комбінація публічної та приватної хмар, що дозволяє обмінюватися даними та застосунками між ними.

Такі хмари можуть використовуватись в залежності від потреб користувачів та характеру бізнес-задач.

Основні функції хмарних обчислень в IoT

- **Зберігання та управління даними.** Забезпечення зберігання великого об'єму даних та зручного доступу до них, а також їх обробка та аналіз в хмарі.
- **Машинне навчання та аналітика.** Використання алгоритмів машинного навчання та аналітичних інструментів для отримання нових інсайтів, покращення прийняття рішень, прогнозування подій та оптимізації роботи системи.
- **Безпека та захист даних.** Забезпечення високого рівня безпеки при передачі, зберіганні та обробці даних в хмарі. Захист від несанкціонованого доступу та моніторинг подій.
- **Маштабованість та гнучкість.** Здатність пристосовуватись до зростаючого обсягу пристроїв IoT та даних, що від них надходять.
- **Віддалене управління та моніторинг.** Забезпечення можливості управління, віддаленої діагностики та налаштування з будь-якої точки світу. Наявність служби технічної підтримки.
- **Інтеграція та стандартизація.** Забезпечення сумісності та легкості інтеграції різноманітних пристроїв та підтримка багатьох стандартів комунікації в IoT.
- **Енергоефективність**. Витрати ресурсів на складні розрахунки, роботу з великими об'ємами даних та інше бере на себе хмара.

Сценарії використання хмар у вбудованих системах

Вбудовані системи можуть вигідно використовувати хмарні технології для поліпшення власної ефективності та розширення функціональності. Розглянемо деякі сценарії та очікувані переваги від цього:

- Розумні пристрої або автомобільні системи, можуть отримувати оновлення програмного забезпечення через хмару. *Переваги*: зменшення необхідності фізичної заміни апаратного забезпечення, швидке розгортання оновлень та підвищення безпеки.
- Збір, аналіз та візуалізація даних з великої кількості сенсорів. *Переваги*: можливість масштабування для обробки великого обсягу даних, віддалений доступ до результатів та використання інтелектуальних аналітичних інструментів.
- Автоматичне резервне копіювання даних в хмарному сховищі для подальшого відновлення. *Переваги*: забезпечення безпеки даних, можливість відновлення у випадку втрати або пошкодження інформації.

- Віддалений доступ до ресурсів вбудованих систем для розробників, тестерів або служби підтримки через хмарні сервіси. *Переваги*: зручність у віддаленому адмініструванні, відладці та підтримці, навіть якщо пристрій знаходиться в іншому місті.
- Забезпечення віддаленого керування параметрами вбудованих систем через хмарні інтерфейси. *Переваги*: можливість швидкого реагування на зміни, оптимізація роботи та підвищення ефективності.
- Інтеграція вбудованих систем у великі мережі для обміну даними та взаємодії з іншими пристроями та хмарними платформами. *Переваги*: розширення можливостей комунікації та обробки інформації, створення інтелектуальних, взаємодіючих між собою, систем.

Постачальники хмарних послуг

Постачальники хмарних послуг (Cloud Providers) – це компанії, які надають доступ до інфраструктури хмарних обчислень та різноманітних послуг через Інтернет. Постачальники дозволяють користувачам отримати доступ до потужних ресурсів хмарної інфраструктури за плату або безкоштовно, спрощуючи розгортання та керування інфраструктурою, а також надаючи широкий спектр інструментів та сервісів.

Популярні постачальники хмарних послуг:

- Amazon Web Services (AWS) – найбільший постачальник хмарних послуг, який надає широкий спектр сервісів та відзначається високою масштабованістю.
- Microsoft Azure – інтегрована хмарна платформа, яка вирізняється сильними інтеграційними можливостями з продуктами Microsoft та широким спектром сервісів для побудови різноманітних додатків.
- Google Cloud – забезпечує потужні сервіси штучного інтелекту та машинного навчання, важливий акцент на аналізі даних.
- IBM Cloud – постачальник хмарних послуг, що вирізняється значними ресурсами для обчислень та великим спектром рішень для підприємств.
- Oracle Cloud – фокусується на інтеграції зі стандартними продуктами та рішеннями для корпоративного сектору, пропонуючи широкий спектр хмарних сервісів.
- Alibaba Cloud – переважно активний в азійському регіоні, відзначається високою доступністю, широким портфелем послуг та зростаючим впливом на глобальному ринку.

Постачальники надають спеціалізовані платформи, що допомагають вирішувати завдання, пов'язані з розробкою, впровадженням та управлінням Інтернетом речей. Деякі з них спрямовані на конкретні галузі або функції. Нижче наведено декілька популярних платформ:

1) AWS IoT Core – дозволяє підключати пристрої, здійснювати автентифікацію, комунікацію та керувати ними. Крім того, AWS надає такі послуги, як AWS IoT Analytics, AWS IoT Device Defender та інші.
   Комунікаційні протоколи: HTTP, MQTT, WebSockets.
   Використання: розумне місто/дім, сільське господарство.

2) Azure IoT Central та IoT Hub – це ключові складові платформи для IoT від Microsoft. Azure IoT Hub забезпечує безпечне підключення та комунікацію з підключеними пристроями, а Azure IoT Central надає інструменти для розробки та керування додатками IoT.
   Комунікаційні протоколи: MQTT, AMQP, WebSockets, HTTPS.
   Використання: виробництво, торгівля, охорона здоров'я.

3) IoT Intelligent Applications – частина інфраструктури хмарних обчислень від Oracle, що використовуючи дані з датчиків підключених пристроїв надає користувачам більше видимості, інсайтів та інтеграції даних.
   Комунікаційні протоколи: HTTP, MQTT.
   Використання: розумне виробництво, обслуговування, логістика, моніторинг безпеки.

4) Bosch IoT Suite – пропонує повнофункціональні попередньо налаштовані інструменти та рішення, що спрощують процес розгортання та використання IoT застосунків. Має високу доступність, надійність та масштабованість, а також ця платформа відкрита та доволі гнучка.
   Комунікаційні протоколи: HTTP, MQTT.
   Використання: енергетика, виробництво, господарство, розумні будівлі.

5) Arduino Cloud – це хмарна платформа, яка надає інфраструктуру для підключення та управління пристроями на основі платформи Arduino. Ця платформа дозволяє розробникам легко взаємодіяти з ними через веб-інтерфейс або мобільний додаток.
   Комунікаційні протоколи: MQTT.
   Використання: розумні будинки, сільське господарство, навчальні проекти.

6) Blynk – це платформа, що спрощує створення з'єднання між пристроями IoT та мобільними пристроями, дозволяючи розробникам створювати інтерактивні електронні пристрої та додатки без глибоких знань в галузі програмування або електроніки.
   Комунікаційні протоколи: використовує власний бінарний протокол Blynk Protocol.
   Використання: розумні будинки, системи вентиляції, обігріву та інше.

7) Dweet.io – це безкоштовна хмарна публічна платформа для обміну даними між пристроями IoT через простий API. Основна ідея полягає в можливості легко надсилати, зберігати та отримувати дані з підключених пристроїв.
   Комунікаційні протоколи: HTTP.
   Використання: публічний обмін повідомленнями між пристроями IoT.

8) ThingsBoard – це відкрита платформа для інтернету речей, яка дозволяє підключати пристрої до хмари та збирати, обробляти і візуалізувати дані з них. Платформа дозволяє створювати власні дашборди (панелі приладів) для моніторингу та взаємодії з даними в режимі реального часу. Також пропонується на вибір декілька способів автентифікації пристроїв для забезпечення безпеки та контролю над доступом.
   Комунікаційні протоколи: MQTT, CoAP, HTTP.
   Використання: моніторинг навколишнього середовища, розумний офіс, відстеження автопарку, енергетика та багато іншого.

Архітектура AWS IoT

Архітектура AWS IoT [15] включає:

- AWS IoT Core – серце платформи, що забезпечує безпечне підключення та управління пристроями без необхідності виділення серверів.
- AWS Greengrass – допомагає керувати застосунками IoT на периферії, обробляти дані в локальній мережі, зберігати їх на тривалий час та інше.
- AWS Kinesis – відповідає за безпечну роботу з відео-потоками.
- AWS IoT ExpressLink – дозволяє швидко перетворити будь-яку вбудовану систему на систему IoT.
- AWS IoT Analytics – інструменти для аналізу та візуалізації IoT-даних.
- AWS IoT Device Defender – допомагає підвищити безпеку пристроїв IoT.

Детальніще про інші цікаві сервіси можна дізнатись на офіційному сайті AWS в розділі IoT [15]. Приклад потоку даних:

1) Пристрої підключаються до AWS IoT Core і надсилають дані.
2) Дані проходять через AWS Greengrass для локальної обробки або надсилаються безпосередньо до AWS Lambda для обробки у реальному часі.
3) Оброблені дані зберігаються у Amazon S3 або DynamoDB.
4) Дані аналізуються за допомогою AWS IoT Analytics і візуалізуються за допомогою AWS QuickSight.

Архітектура Microsoft Azure IoT

Архітектура Microsoft Azure IoT [16] включає:

- Azure IoT Hub – центральний сервіс для управління підключенням IoT-пристроїв та маштабування їх кількості.
- Azure IoT Central – керована платформа, що забезпечує швидке створення, управління та моніторинг IoT рішень без необхідності значних інвестицій у розробку.
- Azure IoT Edge – розширення, що дозволяє виконувати обчислення на периферії.
- Azure Digital Twins – платформа для створення цифрових копій фізичних об'єктів.
- Azure RTOS – операційна система реального часу для вбудованих пристроїв, що забезпечує високу продуктивність і надійність.
- Azure Sphere – платформа для створення безпечних IoT-рішень, яка включає захищені мікроконтролери, операційну систему та хмарні сервіси безпеки.

Більш детально про інші цікаві сервіси можна дізнатись на офіційному сайті Microsoft в розділі привяченому продуктам IoT [16]. Приклад потоку даних:

1) Пристрої підключаються до Azure IoT Hub, надсилають дані.
2) Дані можуть бути оброблені на периферії за допомогою Azure IoT Edge або надсилаються до Azure Stream Analytics для обробки в реальному часі.
3) Оброблені дані зберігаються в Azure Cosmos DB або Azure Blob Storage.
4) Дані аналізуються за допомогою Azure Machine Learning або Azure Time Series Insights для аналітики та візуалізації.

Створення програми для взаємодії з хмарною платформою ThingsBoard

Розглянемо приклад, який може бути використаний як основа для зчитування та передачі даних з датчиків у хмару. В якості мікроконтролера виступає ESP32, що надсилає дані по Wi-Fi до хмарної платформи ThingsBoard за протоколом MQTT. Основні кроки включають:

1) Створення облікового запису та вхід на сайт ThingsBoard (https://thingsboard.io/).
2) Створення нового пристрою у ThingsBoard:

   Панель керування –> Devices –> Add device –> Add new device -> «Вказати назву пристрою, наприклад: MyEsp32» –> Add.

3) Написання програми для пристрою розпочинається з підключення необхідних бібліотек:

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

4) Встановлення параметрів підключення до Wi-Fi:

```
const char *ssid = "Назва_точки_доступу";
const char *password = "Пароль_від_неї";
```

5) Налаштування параметрів MQTT. Отримати необхідні дані для підключення до серверу ThingsBoard можна на сайті, в списку пристроїв, натиснувши на потрібний пристрій (MyEsp32). В вікні «Device details», необхідно натиснути Manage Credentials (керування обліковими даними) та обрати потрібний тип даних (MQTT Basic). Після чого – згенерувати Client Id, User Name, Password, натиснути кнопку Save та внести ці дані в код:

```
const char* clientId = "Згенерований_ClientId";
const char* mqttUsername = "Згенерований_UserName";
const char* mqttPassword = "Згенерований_Password";
```

6) В вікні «Device details» натиснувши кнопку Check connectivity можливо переглянути параметри різних протоколів для обміну даними. Необхідно обрати протокол MQTT та скопіювати надані параметри в код:

```
const char* mqttServer = "mqtt.thingsboard.cloud";
int mqttPort = 1883;
```

Можливо перевірити з'єднання з сайтом виконавши команду для обраної операційної системі в терміналі, наприклад, за протоколом HTTP для ОС Windows команда буде мати наступний вигляд:

```
C:\Users\Admin>curl -v -X POST
http://thingsboard.cloud/api/v1/DEVICE_TOKEN/telemetry --header Content-
Type:application/json --data "{temperature:25}"
```

Результатом має бути відповідь, що свідчить про вдалий обмін даними:

```
Note: Unnecessary use of -X or --request, POST is already inferred.
*   Trying 54.208.96.163:80...
* Connected to thingsboard.cloud (54.208.96.163) port 80
> POST /api/v1/DEVICE_TOKEN/telemetry HTTP/1.1
> Host: thingsboard.cloud
```

```
> User-Agent: curl/8.4.0
> Accept: */*
> Content-Type:application/json
> Content-Length: 16
>
< HTTP/1.1 200
< content-length: 0
< date: Tue, 01 Jan 2024 00:00:00 GMT
<
* Connection #0 to host thingsboard.cloud left intact
```

У вікні «Device details» в розділі «Latest telemetry» з'явиться ключ «temperature» з значенням 25.

7) Ініціалізація об'єктів WiFi та MQTT:

```
WiFiClient wifiClient;
PubSubClient client(wifiClient);
```

8) Створення цілочисельної змінної «**i**», яка ініціалізується значенням **1**, щоб в подальшому, змінюючи її, імітувати дані отримані з давача:

```
int i = 1;
```

9) З'єднання з Wi-Fi та налаштування MQTT відбувається за допомогою функції **setup()** з прикладів у попередніх темах:

```
void setup()
{
  // Ініціалізація порту для виведення даних у консоль
  Serial.begin(115200);
  // Підключення до Wi-Fi
  WiFi.begin(ssid, password);
  Serial.print("Підключення до Wi-Fi...");
  while(WiFi.status() != WL_CONNECTED)
  {
    delay(300);
    Serial.print(".");
  }
  Serial.println("\nПідключення успішне");

  client.setServer(mqttServer, mqttPort);
}
```

10) В основному циклі програми (loop), на початку, перевіряється чи підключено клієнта до MQTT-брокера (ThingsBoard), після чого, значення змінної «**i**» виводиться у консоль.

```
void loop()
{
  if (!client.connected())
  {
    reconnect();
  }
  Serial.println("Поточна температура: " + String(i));
```

Примітка. В наведеному коді використовується конкатенація рядків. **Конкатенація** – це процес об'єднання двох або більше рядків в один. У виразі вище **`"Поточна температура: "`** – це текстовий літерал (рядок), а **`String(i)`** – це конструктор рядка для перетворення значення змінної «**`i`**» (яке може бути числом, символом чи іншим типом даних) в об'єкт рядка (**`String`**). Оператор **`+`** використовується для об'єднання цих двох рядків. В деяких випадках може бути ефективніше використовувати декілька викликів **`Serial.println()`** з окремими параметрами.

Функція **`publish`** в бібліотеці PubSubClient має два обов'язкові параметри – «Топік (**`topic`**)» та «Дані (**`payload`**)». Для них платформа ThingsBoard використовує власний формат. Це робиться для забезпечення стандартизації та легшого розуміння внутрішньої структури даних при розгортанні різних типів пристроїв та додатків на платформі.

- **`"v1/devices/me/telemetry"`** – це топік для надсилання даних на ThingsBoard. Зазвичай використовується структура v1/devices/*{device_id}*/telemetry, де *{device_id}* – ідентифікатор пристрою на ThingsBoard, що автоматично підставляється на місце «me».
- **`payload`** – це рядок, який містить дані (у форматі JSON) про температуру або інші параметри, які відправляються на ThingsBoard. Дані надсилаються у вигляді JSON-структури, оскільки JSON є зручним форматом для обміну структурованими даними, і ThingsBoard легко розпізнає та обробляє дані в цьому форматі. Функція **`c_str()`** конвертує об'єкти рядка типу **`String`** у константний масив символів (C-style string), оскільки другий параметр функції **`publish`** поминен бути саме таким.

```
  String payload = "{\"temperature\":" + String(i) + "}";
  client.publish("v1/devices/me/telemetry", payload.c_str());
```

В кінці циклу loop змінна «**`i`**» збільнується на одиницю, імітуючи збільшення температури. Затримка в 1 секунду введена, для обмеження частоти публікацій та уникнення надмірного трафіку до сервера ThingsBoard.

```
  i++;
  delay(1000);
}
```

11) Функція **`reconnect()`**, на відміну від попереднього прикладу, перед спробою підключитися до ThingsBoard перевіряє підключення до мережі Wi-Fi та виконує перепідключення, за потреби. Виклик **`connect`**, окрім **`clientId`**, потребує додаткових параметрів: **`mqttUsername`** та **`mqttPassword`**, для виконання аутентифікації у хмарі:

```
void reconnect()
{
  //Перевірка і перепідключення до Wi-Fi
  while (!client.connected())
  {
    if (WiFi.status() != WL_CONNECTED)
    {
      Serial.print("Підключення до Wi-Fi...");
```

```
      WiFi.begin(ssid, password);
      while(WiFi.status() != WL_CONNECTED)
      {
        delay(300);
        Serial.print(".");
      }
      Serial.println("Підключено");
    }

    //Перепідключення до хмари
    Serial.println("Підключення до ThingsBoard...");
    if (client.connect(clientId, mqttUsername, mqttPassword))
    {
      Serial.println("Клієнт: " + String(clientId) + " підключений.");
    }
    else
    {
      Serial.println("помилка: " + String(client.state()) + " повторна спроба
через 3 секунди.");
      delay(3000);
    }
  }
}
```

12) Запуск пристрою. Завантаживши вищенаведений код в мікроконтролер, за допомогою повідомлень у консолі, можливо переконатися, що:
    - Пристрій підключається до мережі Wi-Fi:
      ```
      Підключення до Wi-Fi......
      Підключення успішне
      ```
    - З'єднання з сервером ThingsBoard встановлено.
      ```
      Підключення до ThingsBoard...
      Клієнт: {device_id} підключений.
      ```
    - Значення температури відправляється на сервер та зростає:
      ```
      Поточна температура: 1
      Поточна температура: 2
      Поточна температура: 3
      ```
13) Перевірка на сервері. Для перевірки останніх телеметричних даних на сервері ThingsBoard необхідно:
    - У меню «Devices» обрати пристрій, який надсилає дані на сервер (MyESP32).
    - У вікні «Device details» перейти на вкладку «Latest Telemetry» (Остання телеметрія). При надходженні від пристрою нових телеметричних даних, вони будуть відображатися тут.
    - Знайти рядок з ключем «temperature» та перевірити, що його значення автоматично оновлюється (зростає) щосекунди.
14) Створення панелі приладів (Dashboard). Платформа ThingsBoard надає можливість моніторингу показників на панелі приладів, посилання на яку можна навіть зробити публічним.

Панель приладів IoT (Dashboard)

Панелі приладів IoT (Dashboard) – це інструмент візуалізації, який дозволяє користувачам спостерігати за станом пристроїв, аналізувати дані та приймати рішення на основі отриманої інформації. Зазвичай графічний інтерфейс панелі відображає різні параметри та дані з датчиків в реальному часі. Хмарні платформи для IoT-застосунків дозволяють створювати потужні дашборди з різноманітними віджетами.

Для створення панелі приладів на платформі ThingsBoard, слід виконати наступні кроки:

1) Після входу в систему перейти у розділ «Dashboards».
2) Натиснути на кнопку «Add dashboard» та обрати «Create new dashboard». Також можна імпортувати існуючу панель.
3) Задати ім'я та інші налаштування для нового дашборду.
4) Визначити віджети (візуальні елементи), які потрібно використовувати на дашборді та додати їх, наприклад:
    - Графіки (Time series chart) – відображення змін значень параметрів з часом.
    - Картки (Card) – відображення текстової інформації або значень параметрів.
    - Тривоги (Alarm) – відображення стану або подій в реальному часі з можливістю налаштування візуальних попереджень.
    - Таблиці (Tables) – організація відображення табличних даних.
    - Лічильники (Count widgets) – відображення та оновлення кількості певних подій чи параметрів.
    - Мапа (Map) – відображення місцезнаходження пристроїв на мапі.
    - Аналогові вимірювальні прилади (Analogue gauges) – візуальне відображення значеннь на шкалі.
    - Керування (Control widgets) – відправка команд та управління пристроями.
    - Індикатори (Status indicators) – показують поточний стан пристроїв або параметрів.
    - Промислові віджети (Industrial widgets) надають можливості для відображення даних, специфічних для промислових систем.
    - Середовище (Indoor/Outdoor environment) – відображення параметрів навколишнього середовища, таких як температура чи вологість.
    - Рівень рідини (Liquid level) – відображення рівня рідини у контейнерах.
    - Цифрові вимірювальні прилади (Digital gauges) – візуалізують значення параметрів у цифровій формі.
    - Якість повітря (Air quality) – відображення даних про склад повітря.
    - Введення (Input widgets) – дозволяють отримувати вхідні дані від користувача.
    - Входи/виходи (GPIO) – взаємодія з входами/виходами мікроконтролера.
5) Налаштувати параметри віджета, вказавши, як та які дані він повинен відображати.
6) Визначити джерело даних, обравши зі списку потрібний пристрій та вказавши «Data key». В наведеному вище прикладі ключ лише один – це `temperature`.
7) Збереження та перегляд дашборду.

Використання дашбордів полегшує взаємодію з системою або пристроєм, роблячи інформацію зрозумілою та легкодоступною для користувача.

Запитання для самоперевірки

1. Що таке великі дані?
2. Як визначити що дані дійсно великі?
3. Як можна класифікувати дані?
4. Чим відрізняються структуровані та неструктуровані дані?
5. Скільки даних збирають давачі та як забезпечити їх зберігання в сучасному світі?
6. Як називають систему з багатьох серверів, розміщених в різних центрах обробки даних?
7. Які переваги використання хмарних сервісів для окремо взятої людини та підприємства?
8. Назвіть основні моделі "хмари" та розкажіть про їх особливості.
9. Які три моделі хмарних послуг можна виділити, в чому їх відмінності?
10. Для чого аналізуваи дані та як це можна робити?
11. Назвіть популярних постачальників хмарних послуг для IoT.
12. Які протоколи прикладного рівня найбільш популярні для взаємодії з хмарними платформами?

Тести

1. Що з наведеного відноситься до Big Data?
   A) Обсяг.
   B) Різноманітність.
   C) Швидкість.
   D) Усі варіанти вірні.
2. Яка характеристика найбільше підходить для даних, що були отримані з давачів, веб-серверів та електронних документів?
   A) Однорідність.
   B) Обсяг.
   C) В’язкість.
   D) Різноманітність.
3. За збереження даних на декількох серверах відповідає:
   A) Система керування базою даних.
   B) Граничне обчислення.
   C) Розподілена обробка даних.
   D) MySQL.
4. Що з наведеного безпосередньо генерує великі обсяги даних?
   A) Розподілені обчислення.
   B) Давачі.
   C) Граничні пристрої.
   D) Бази даних.
5. За що зазвичай відповідають пристрої локальної мережі, перед відправкою даних до глобальної мережі?
   A) Граничні обчислення.
   B) Зменшення очікування.
   C) Підвищення швидкості.
   D) Збереження в хмарі.

6. Неструктуровані дані:
   A) Мають чітку структуру та організацію у вигляді таблиць або баз даних.
   B) Зберігаються у форматі, що може бути представлений у різних формах.
   C) Не мають чіткої організації, наприклад – аудіо- та відео-записи.
   D) Містяться в реляційних базах даних.
7. Легкий формат обміну даними, який базується на синтаксисі мови JavaScript, де дані записуються у вигляді тексту це:
   A) HTTP.
   B) JSON.
   C) ZIP.
   D) WWW.
8. Дані, які мають чітко визначену організацію, такі як таблиці або бази даних це:
   A) Лінійні дані.
   B) Структуровані дані.
   C) Неструктуровані дані.
   D) HTML.
9. Яка модель хмарних обчислень допомагає знизити витрати на інфраструктуру, але також знижує і контроль за власними даними?
   A) Публічна хмара.
   B) Гібридна хмара.
   C) Хмара спільноти.
   D) Приватна хмара.
10. У якій моделі за ідентифікацією користувачів відповідає клієнт хмарних послуг, а не їх провайдер?
   A) Інфраструктура як сервіс (IaaS).
   B) Платформа як сервіс (PaaS).
   C) Програмне забезпечення як сервіс (SaaS).

ТЕМА 7. ОПЕРАЦІЙНІ СИСТЕМИ РЕАЛЬНОГО ЧАСУ (RTOS)

План лекції

- Типи операційних систем (GPOS vs RTOS)
- Типи RTOS
- Багатозадачність
- Приклад застосування FreeRTOS

Текстова частина лекції

Операційні системи

Операційна система (ОС) – це своєрідний "керівник" пристрою, який виконує різні важливі завдання, щоб забезпечити його надійну та ефективну роботу. ОС управляє виконанням програм, розподіляє обчислювальну потужність та пам'ять, забезпечує надійність та стабільність роботи пристрою.

Використання операційної системи надає програмістам численні переваги:

- **Спрощення взаємодії з обладнанням**. Абстракція від конкретного обладнання дозволяє програмістам працювати на високому рівні без необхідності детального розуміння апаратної частини пристрою.
- **Підтримка драйверів та периферійних пристроїв**. Стандартизований інтерфейс для взаємодії з різними пристроями полегшує розробку драйверів та інтеграцію нового обладнання.
- **Багатозадачність**. Можливість виконання кількох завдань одночасно полегшує розробку складних програм і підвищує продуктивність.
- **Управління ресурсами**. ОС розподіляє ресурси (процесор, пам'ять, ввід/вивід) між різними програмами, що дозволяє ефективно використовувати обладнання.
- **Стандартізація**. Стандарти ОС для роботи з різними аспектами пристроїв дозволяють використовувати написані програми на різних платформах.
- **Безпека та захист**. Забезпечення заходів безпеки, таких як контроль доступу, вірусні сканери, файерволи та інші, що допомагають у захисті від різноманітних загроз.

У вбудованих системах використовуються два основних типи операційних систем:

- **Операційні системи загального призначення (GPOS, General Purpose Operating System)** – використовуються у різних сценаріях управління великою кількістю процесів та виконання багатьох завдань за одиницю часу, де немає жорстких вимог до виконання в точно визначений час. Для забезпечення доступу до CPU різним процесам і задачам GPOS використовують політику розподілу ресурсів. Забезпечення великої продуктивності відбувається шляхом виконання якумога більшої кількості завдань, не дивлячись на їх важливість, що може спричинити затримку виконання для дійсно важливих завдань.
- **Операційні системи реального часу (RTOS, Real-time Operating System)** гарантують, що виконання операцій та завдань відбувається за строго визначеними часовими рамками. Це дозволяє передбачати та керувати часовою поведінкою системи. Система

готова негайно реагувати на події та виконувати відповідні завдання з найменшими можливими затримками. В RTOS завдання може мати визначений пріоритет, і система віддає перевагу виконанню важливіших завдань перед менш важливими.

Вибір між звичайною операційною системою (GPOS) та операційною системою реального часу (RTOS) залежить від конкретних вимог та характеристик системи, що розробляється.

Рекомендації для застосування RTOS:

- Потребується невелика та легка ОС для полегшення взаємодії з апаратною частиною.
- Вбудована система має жорсткі вимоги до часу виконання завдань і вимагає гарантованої відповіді на події у визначений термін.
- В системі присутні багато завдань і важливо забезпечити правильну взаємодію та синхронізацію між ними.
- Вбудована система сумісна з RTOS.

Архітектура Super Loop (супер петля/цикл)

Super Loop – це архітектура програмування, яка часто використовується у вбудованих системах з обмеженими ресурсами. Програма організована у вигляді безперервного циклу, що виконує всі необхідні завдання послідовно.

Основні характеристики:

- **Безперервний цикл.** Програма постійно повторює цикл виконання завдань.
- **Послідовність виконання.** Завдання виконуються одне за одним у визначеному порядку.
- **Простота реалізації.** Легко зрозуміти і реалізувати для невеликих систем.

***Переваги*:**

- **Мінімальні надлишкові витрати**. Не потребує складного керування задачами.
- **Прогнозованість**. Виконання завдань завжди відбувається в тому самому порядку.

***Недоліки*:**

- **Обмежена багатозадачність**. Всі завдання виконуються послідовно.
- **Часова нестабільність**. Якщо одне завдання займає більше часу, ніж очікувалося, це впливає на всі інші завдання в циклі.
- **Складність у масштабуванні**. Додавання нових завдань може значно збільшити час виконання циклу.

Багатозадачність

Для складних систем, де паралельно повинна виконуватись велика кількість задач та забезпечуватись детермінованість і гнучкість управління, краще використовувати операційні системи реального часу (RTOS). Програми для RTOS складаються з окремих завданнь, які виконуються незалежно одне від одного.

Завдання (Task) – це основна одиниця виконання, яка представляє незалежну функцію, що виконує певну задачу в системі.

Замість одного супер-циклу – кожне завдання виконується в окремому, своєму, циклі. Код кожного завдання може бути не пов’язаний з будь-яким іншим. Це значно спрощує процес розробки, а за різні завдання можуть відповідати різні розробники.

В ідеалі, для паралельного виконання багатьох завдань, за кожне з них повинне відповідати окреме ядро процесора, але, RTOS дозволяє створити ілюзію паралельного виконання багатьох завдань навіть на одному ядрі. Це досягається за допомогою механізму багатозадачності, при якому ОС швидко перемикається між завданнями (context switching), виконуючи по черзі невеликі частинки кожного з них. При таких перемикання постійно виконуються дві речі:

- **Збереження контексту**. Поточне завдання зберігає свій контекст (регістри процесора, програмний лічильник, стек) так, щоб після повернення до нього, воно могло продовжити виконання з того самого місця.
- **Відновлення контексту**. Нове завдання відновлює свій контекст і продовжує виконання з того місця, де воно зупинилося.

Архітектура RTOS

Архітектура RTOS може бути розділена на декілька основних компонентів.

Ядро (kernel) – це основна частина RTOS, яка відповідає за планування завдань, управління ресурсами та обробку переривань. Саме воно визначає, яке завдання буде виконуватися в конкретний момент часу. Ядро вбудоване в операційну систему і діє як її центральна частина, що реалізує забезпечення вимог реального часу та ефективну і надійну роботу.

Крім ядра, RTOS може включати модулі для мережевого взаємодії, синхронізації процесів, а також інші додаткові служби. Модульна структура дозволяє вбудованим системам використовувати лише ті частини, що необхідні для конкретного застосування.

Основні функції ядра включають:

- **Управління завданнями,** створення, запуск, зупинка та блокування. Відстеження статусу та пріоритету кожного завдання, а при готовності до виконання декількох завдань – вибір відповідно до плану.
- **Планування завдань** – визначає порядок виконання завдань в системі за допомогою алгоритмів планування, які враховують пріоритети, часові обмеження, залежності та інші фактори. Саме планування дозволяє ефективно використовувати обчислювальні ресурси та гарантує виконання завдань у визначений час.
- **Синхронізація завдань** – дозволяє координувати роботу різних завдань, які можуть взаємодіяти між собою або зі спільними ресурсами. Можуть використовуватись різні механізми, такі як семафори, м’ютекси, події, черги, таймери, сигнали та інші. Завдяки ним, ядро гарантує, що тільки одне завдання може використовувати спільний ресурс у певний момент.
- **Управління пам'яттю** – відповідає за виділення та звільнення пам’яті для завдань. Можна виділити два типи: Stack та Heap.
- **Управління часом** – це точне вимірювання, відлік та порівняння часу для планування, синхронізації та виконання завдань. Для цього може використовуватися системний годинник, переривання, таймери, лічильники тощо.

- **Обробка переривань** – дозволяє системі реагувати на асинхронні події, які виникають поза ядром, через переривання. Наприклад, на введення інформації користувачем чи сигнали від пристроїв.
- **Керування вводом/виводом пристрою** – забезпечує обмін даними між системою та зовнішніми пристроями через драйвери вводу/виводу.

Планувальник

Для виконання декількох задач час роботи процесора поділяється на невеликі інтервали – **тактові інтервали процесора** (**ticks**, **тіки**), наприклад, по одній мілісекунді. Такий поділ ще називається – **часовий розріз**. Кожну мілісекунду апаратний таймер перериває процесор та викликається планувальник, що вирішує якому завданню буде присвячений наступний тік. Для цього планувальник може застосовувати декілька підходів, а саме:

- **Пріоритетне планування** – завдання з вищим пріоритетом виконуються першими.
- **Round-Robin** – завдання з однаковим пріоритетом виконуються по черзі, отримуючи однакові часові інтервали.
- **Планування на основі подій** – завдання виконуються у відповідь на певні події або сигнали.

В процесі виконання програми завдання можуть перебувати у різних станах, які визначають їх поведінку та подальше планування. У системі FreeRTOS це [17]:

- **Ready** (Готове до виконання). Завдання отримує цей статус одразу після створення (`xTaskCreate`). Така завдання готове до виконання, але не виконується в поточний момент часу через те, що процесор зайнятий іншим завданням з таким же або вищим пріоритетом. Коли процесор стане доступним або після завершення блокування/затримки (`vTaskDelay`, `xQueueReceive`), завдання в цьому стані буде вибрано для виконання і планувальник вирішить його виконати.
- **Running** (Виконується). Завдання виконується в поточний момент часу. В системі з одним ядром лише одне завдання може перебувати в цьому стані.
- **Blocked** (Заблоковане). Завдання чекає на певну подію (наприклад, затримку часу, сигнал від іншого завдання або звільнення ресурсу). Таке завдання переходить в стан Ready, коли очікувана подія відбудеться. Функції для блокування:
  - `vTaskDelay(ticks)` – завдання чекає задану кількість тактів;
  - `vTaskDelayUntil(lastWakeTime, period)` – завдання чекає до певного моменту часу;
  - `xQueueReceive(queue, buffer, ticksToWait)` – завдання чекає на отримання повідомлення з черги;
  - `xSemaphoreTake(semaphore, ticksToWait)` – завдання чекає на отримання семафора.
- **Suspended** (Призупинене). Завдання призупинено і не виконується, поки його явно не відновлять (Resumed). Функції для призупинення та відновлення:
  - `vTaskSuspend(taskHandle)` – призупиняє завдання;
  - `vTaskResume(taskHandle)` – відновлює призупинене завдання;

Наприклад, припустимо, є три завдання з різними пріорітетами:

- **Task1** (Пріоритет 1 – найменший).
- **Task2** (Пріоритет 2 – середній).
- **Task3** (Пріоритет 3 – найбільший).

При запуску планувальника він робить наступні дії:

1) Обирає завдання з найбільшим пріоритетом, готове до виконання. У цьому випадку це Task3.
2) Після завершення виконання Task3 або при його блокуванні (наприклад, через виклик **`vTaskDelay()`**), планувальник переходить до наступного завдання з найбільшим доступним пріоритетом, тобто Task2.
3) Якщо всі завдання з більшими пріоритетами заблоковані або завершені, планувальник обирає Task1.

Якщо Task3 стає готовим (наприклад, закінчується затримка) під час виконання Task1 або Task2, планувальник негайно перемикається на Task3, оскільки воно має найвищий пріоритет.

Потреба у системах реального часу

Вбудовані системи реального часу (RTES, Real-time embedded systems) є спеціальним типом обчислювальних систем, призначених для виконання конкретних функцій у реальному часі. Вони використовуються у різних галузях, таких як автомобільна промисловість, медицина, промисловий контроль, телекомунікації та багато інших, де важливо, щоб завдання виконувалися точно та вчасно, наприклад:

- **Автомобільні системи.** Для контролю за антиблокувальною системою гальм, системою стабілізації, подушками безпеки та іншим. RTOS дозволяє системам реагувати на зміни в умовах руху автомобіля миттєво та точно.
- **Медичні пристрої.** Наприклад, в кардіостимуляторах, які використовуються для регулювання серцевого ритму та надають стимуляцію серцевому м'язу за необхідності. Система реального часу може ефективно обробляти події, такі як зміни серцевого ритму і призначати відповідні заходи.
- **Промислове автоматизоване обладнання**. У промисловості, де необхідно забезпечити синхронізовану роботу багатьох пристроїв, RTOS використовуються для точного управління обладнанням.
- **Робототехніка**. У робототехніці RTOS може використовуватися для управління рухами роботів та реагування на зміни в навколишньому середовищі в режимі реального часу.

В усіх цих прикладах RTOS гарантує, що система може ефективно та надійно виконувати свої функції у визначені терміни, що є критично важливим для правильної роботи пристроїв у реальному світі.

Системи м’якого та жорсткого реального часу

RTOS можна розділити на два типи:

- М'якого реального часу (М’які RTOS).
- Жорсткого реального часу (Жорсткі RTOS).

Вони відрізняються за рівнем жорсткості виконання часових обмежень та наслідків порушення цих обмежень.

Жорсткі RTOS – гарантують, що завдання буде виконано точно та вчасно. Найважливіша характеристика – неприпустимість порушення часових обмежень. В ситуаціях, коли завдання не виконується вчасно, продовження його виконання може призвести до подальших проблем, тому воно зупиняється і блокується. В основному такі системи використовуються в критичних системах, де порушення термінів виконання може призвести до серйозних наслідків, наприклад, в авіоніці чи медичному обладнанні.

М'які RTOS – надають більшу гнучкість щодо часових обмежень, дозволяючи пропускати деякі терміни виконання без критичних наслідків. Вони забезпечують високу ймовірність вчасного виконання, але не гарантують його. Застосовуються в випадках, де втрата термінів може призвести до погіршення якості обслуговування, наприклад, мультимедійні системи, де втрата деякої кількості кадрів або аудіосемплів не призводить до критичних наслідків.

Перевагою систем з жорстким реальним часом є гарантія вчасного виконання завдань, систем з м’яким реальним часом – більш ефективне використання ресурсів, що призводить до вищої продуктивності.

Популярні RTOS

Кількість доступних операційних систем надзвичайно велика. Більше ста варіантів RTOS наведено у таблиці за посиланням [18]. Вони класифікуються за наступними критеріями:

1) **Ліцензія**. RTOS можуть мати вільну або пропрієтарну ліцензію. Пропріетарні ОС мають обмеження в використанні та розповсюдженні і належать певному виробнику. Вільні – без обмежень доступні для використання, модифікації та розповсюдження.
2) **Модель вихідного коду**. ОС може мати відкритий або закритий вихідний код. Закритий код означає, що він належить конкретному виробнику і доступний лише йому, а клієнти мають доступ лише до відкомпільованої програми. Відкритий код доступний для спільної розробки та вдосконалення спільнотою.
3) **Цільове використання**. Залежно від того, чи оптимізована RTOS для роботи на певному апаратному забезпеченні, з обмеженими ресурсами, високими вимогами до продуктивності і надійності, існують RTOS як для вбудованого використання, так і для систем загального призначення.
4) **Статус**. Система може бути «активною» або мати інший статус, залежно від того, чи продовжується її розробка та підтримка. Також можна зустріти наступні статуси систем: dormant (спляча); defunct (недіюча); maintenance only (тільки обслуговування).

5) **Платформи**. Залежно від архітектур процесорів RTOS може підтримувати різні платформи, наприклад, FreeRTOS підтримує: ARM, AVR, AVR32, ColdFire, ESP32, HCS12, IA-32, Cortex-M3-M4-M7, Infineon XMC4000, MicroBlaze, MSP430, PIC, PIC32, Renesas H8/S, RISC-V, RX100-200-600-700, 8052, STM32, TriCore, EFM32.

Крім цього, важливим фактором є розмір ОС, який зазвичай залежить від набору функціональностей. Маленькі ОС можуть мати мінімальний набір функцій (планувальник, механізми синхронізації та зв'язку) та займати невеликий обсяг пам'яті (від декількох кілобайт), що робить їх ідеальними для пристроїв з обмеженими ресурсами. З додаванням функціональностей розміри систем зростають. Великими вважаються системи розміром від мегабайту, які можуть підтримувати різноманітні складні функціональності (робота з мережею, графічні інтерфейси, робота з файловими системами і дисками та інше).

Приклади маленьких RTOS:

- **FreeRTOS** – це провідна операційна система на ринку. Вона повністю вільна та відкрита, а ядро FreeRTOS вважається стандартом для MCU. Вона підтримує багатозадачність, програмні таймери, різні бібліотеки, платформи та архітектури. В той же час, вона є доволі простою, має невеликий розмір і споживання енергії.
- **TinyOS** – вбудована операційна система для пристроїв невеликої потужності. Вона доволі ефективна для I/O-орієнтованих додатків, але виникають труднощі з інтенсивними задачами.
- **Contiki** – фокусується на малопотужних пристроях з обмеженим об'ємом пам'яті. Система має вбудовані мережеві механізми для стеку TCP/IP, які підтримують IPv4 та IPv6. Розмір системи може варіюватися, але, для багатозадачності з підтримкою TCP/IP їй потрібно лише близько 10 кілобайт RAM та 30 кілобайт ROM.

Приклади великих RTOS (їх розміри можуть значно варіюватися в залежності від конкретної реалізації та конфігурації):

- **RTLinux**, **Real-time Linux** та інші. Стандартне ядро Linux не може гарантувати мінімальну затримку при обробці переривань та переході між завданнями і не підходить для реального часу. Однак, існують різні способи адаптації Linux для реального часу, наприклад деякі RTOS виконують Linux, як одну із задач. Також, існують патчі, що додають можливість повної преемптивності ядру Linux.
- **Windows (Embedded, IoT, CE)** – різні версії операційної системи Windows, спеціально адаптовані для вбудованих систем, таких як касові апарати, банкомати, засоби самообслуговування тощо. Існують версії як RTOS, так і GPOS.
- **BlackBerry QNX** – надає багато переваг для вбудованих систем, які вимагають високої надійності, продуктивності і безпеки. Вона має мікроядерну архітектуру, що забезпечує високу стійкість до збоїв. Система сертифікована за різноманітними стандартами безпеки.

Встановлення та використання FreeRTOS

ESP32, за замовчуванням, вже має FreeRTOS, тому додатково підключати його не потрібно. Для використання FreeRTOS на інших мікроконтролерах, систему слід правильно встановити (підключити), для цього потрібно:

- Завантажити вихідний код системи з офіційного сайту freertos.org.
- Розпакувати завантажений архів, що включає дві основні підпапки: FreeRTOS та FreeRTOS-Plus. Перша містить ядро RTOS та демонстраційні проекти для різних мікроконтролерів, а друга – додаткові бібліотеки та функції для IoT-застосунків. Також, в завантажених файлах можна знайти посилання на інструкції для роботи з FreeRTOS.
- Відкрити FreeRTOS/Demo і знайти папку, яка відповідає потрібному мікроконтролеру та середовищу розробки.
- Додати файли з вихідним кодом FreeRTOS до власного проекту.
- За потреби, у файлі FreeRTOSConfig.h, налаштувати параметри RTOS, дотримуючись інструкцій документації.
- Зібрати проект та завантажити його у мікроконтролер. Перевірити як працює демонстраційна програма, яка запускає декілька задач.

Детально ознайомитись з функціями, які використовуються для роботи з FreeRTOS, можна за допомогою офіційної документації, де, наприклад, описано:

- Функції для створення, запуску, призупинення, видалення та інших операцій з задачами: **`xTaskCreate`**, **`vTaskStartScheduler`**, **`vTaskDelay`**, **`vTaskDelete`**.
- Функції для операцій з семафорами та м’ютексами, які використовуються для синхронізації доступу до спільних ресурсів: **`xSemaphoreCreateBinary`**, **`xSemaphoreTake`**, **`xSemaphoreGive`**.
- Функції для операцій з чергами, які використовуються для комунікації між задачами: **`xQueueCreate`**, **`xQueueSend`**, **`xQueueReceive`**.
- Функції для створення, запуску, зупинки та інших операцій з таймерами, які використовуються для виклику функцій через певний час або з певною періодичністю: **`xTimerCreate`**, **`xTimerStart`**, **`xTimerStop`**.
- Функції для отримання інформації про стан системи, такі як **`vTaskGetRunTimeStats`**, **`xTaskGetTickCount`**, **`uxTaskGetStackHighWaterMark`**.

Приклад програми для паралельного виконання декількох задач

Наведена далі програма демонструє паралельну роботу двох завдань, а саме: миготіння світлодіодом та відслідковування натискань кнопки, що вмикає інший світлодіод.

Основні частини програми:

1) Оголошення пінів та змінної:

```
#define PIN_LED1 19
#define PIN_LED2 33
#define PIN_BUTTON 14

int count = 0;
```

Оголошуються піни світлодіодів (**PIN_LED1** та **PIN_LED2**) та кнопки (**PIN _BUTTON**), а також змінна **count**, яка використовується для відстеження тривалості натискання кнопки.

2) Функція миготіння першого світлодіода:

```
void ledBlink(void * pvParameters)
{
  while (true)
  {
    digitalWrite(PIN_LED1, HIGH);
    delay(500);
    digitalWrite(PIN_LED1, LOW);
    delay(500);
  }
}
```

У безкінечному циклі з інтервалом 500 мілісекунд перемикається стан світлодіода з **HIGH** в **LOW**. Функція приймає один параметр **pvParameters**, який є вказівником загального типу (**void***), він може використовуватись для передачі будь-яких додаткових даних у функцію (в даному випадку – не використовується).

3) Функція відстеження натискань кнопки:

```
void buttonPressed(void * pvParameters)
{
  while (true)
  {
    if (!digitalRead(PIN_BUTTON))
    {
      digitalWrite(PIN_LED2, HIGH);
      Serial.println("Лічильник: " + String (count));
      count++;
    }
    digitalWrite(PIN_LED2, LOW);
  }
}
```

Безкінечно перевіряє стан кнопки, підключеної до піна **PIN_BUTTON**. Доки кнопка натиснута – вмикає світлодіод на піні **PIN_LED2**, збільшує значення змінної **count** та виводить його у Serial Monitor.

4) Ініціалізація та старт завдань у функції **setup**:

```
void setup()
{
  Serial.begin(115200);
  pinMode(PIN_LED1, OUTPUT);
  pinMode(PIN_LED2, OUTPUT);
  pinMode(PIN_BUTTON, INPUT_PULLUP);

  xTaskCreate(ledBlink, "LedBlink", 1000, NULL, 1, NULL);
  xTaskCreate(buttonPressed, "ButtonDown", 1000, NULL, 2, NULL);
}
```

По-перше, встановлюється взаємозв'язок з Serial Monitor із швидкістю 115200 біт за секунду. Далі, використовуючи функцію **pinMode**, налаштовуються піни введення-виведення для керування світлодіодами та кнопкою: **PIN_LED1** та **PIN_LED2** встановлюються в режим виведення (**OUTPUT**), а **PIN_BUTTON** в режим введення з використанням підтягуючого резистора (**INPUT_PULLUP**), для уникнення непередбачуваних значень на цьому піні.
Далі використовується функція **xTaskCreate** з бібліотеки FreeRTOS для створення та запуску двох завдань: **ledBlink** та **buttonPressed**. Кожне завдання отримує ім'я, розмір стеку (**1000**), параметр **pvParameters** (не використовується, тому **NULL**), пріоритет та дескриптор завдання (в даному випадку також **NULL**). Завдання **ledBlink** встановлено з меншим пріоритетом (**1**), ніж завдання **buttonPressed** (**2**), що визначає порядок їх виконання у випадку конфліктів. Більш детальний опис параметрів функції **xTaskCreate** можна знайти в офіційній документації за посиланням [19].

5) Головний цикл (функція **loop**):

```
void loop()
{
  //Тут нічого не виконується :(
}
```

Цикл **loop()** порожній, оскільки вся логіка виконується у контексті завдань. Які, після створення, виконуються безкінечно. Включити код у цикл **loop()** можливо, але слід розуміти, що він буде викликатися паралельно із іншими завданнями, які плануються FreeRTOS.

Деякі версії мікросхем мають більше одного ядра. Наприклад, в двоядерні версії ESP32, постачальником додано функцію **xTaskCreatePinnedToCore**, яка є розширенням до FreeRTOS. Її можливо використовувати замість функції **xTaskCreate**. Відрізняється вона лише додатковим параметром – **CoreId**, що дозволяє прикріпити завдання до конкретного ядра мікроконтролера.

Слід зазначити, що в наведеному вище коді, для забезпечення паузи, використовується функція **delay()**, однак її використання в контексті багатозадачності, особливо в системі реального часу, може впливати на продуктивність та точність виконання завдань. Тому, у FreeRTOS доцільно використовувати функцію **vTaskDelay()**. Вона змінює стан завдання на «заблоковано» на певну кількість тіків (Ticks – це одиниці часу, які використовуються FreeRTOS для вимірювання затримок та таймерів). Це дозволяє не втрачати процесорний час на «очікування» і дає можливість ефективніше виконувати інші завдання. У FreeRTOS, для перетворення кількості мілісекунд в Ticks, можна використати макрос **pdMS_TO_TICKS**. Тобто, **delay(500)** доцільно замінити на **vTaskDelay(pdMS_TO_TICKS(500))**.

Запитання для самоперевірки

1. Що таке операційна система?
2. В чому відмінність між ОС загального призначення (GPOS) та ОС реального часу (RTOS)?
3. Що таке супер-цикл (Super Loop)?
4. Навіщо потрібна RTOS?

5. Чим відріжняються RTOS жорсткого та м'якого реального часу? Які переваги та недоліки кожної з них?
6. На яких мікроконтролерах та в яких галузях доцільно застосовувати RTOS?
7. Що таке багатозадачність? Яким чином вона досягається на процесорах з одним ядром?
8. За що відповідає планувальник в RTOS?
9. Назвіть декілька основних функцій FreeRTOS.

Тести

1. В чому відмінність між ОС загального призначення (GPOS) та ОС реального часу (RTOS)?
   A) GPOS має менше функцій, ніж RTOS.
   B) RTOS немає можливості взаємодії з користувачем.
   C) RTOS гарантує виконання завдань відповідно до часових обмежень.
   D) GPOS завжди працює швидше за RTOS.
2. В чому особливості RTOS жорсткого реального часу?
   A) Гарантує виконання завдань лише у жорстко визначений термін, але не пізніше.
   B) Має гнучкий графік виконання завдань.
   C) Забезпечує гарантований час відповіді на зовнішні події.
   D) Не має обмежень щодо часу виконання задач.
3. В чому особливості RTOS м'якого реального часу?
   A) Гарантує виконання завдань лише у жорстко визначені терміни, але не пізніше.
   B) Має гнучкий графік виконання завдань.
   C) Може пропускати чи затримувати виконання деяких завдань, якщо це не критично для системи.
   D) Не має обмежень щодо часу виконання задач.
4. За що відповідає планувальник в RTOS?
   A) За створення планів роботи виробництва.
   B) За вибір порядку виконання завдань та їх розподіл на процесорі.
   C) За розподіл вільних ресурсів комп'ютера.
   D) За розміщення програм на жорсткому диску.
5. Навіщо потрібна RTOS?
   A) Для створення графічного інтерфейсу користувача.
   B) Для гарантованого виконання багатьох завдань.
   C) Для роботи з великими обсягами даних.
   D) Для створення мережевих додатків.
6. Назвіть декілька основних функцій FreeRTOS.
   A) Планування завдань, керування потоками, синхронізація, управління пам'яттю.
   B) Створення графічного інтерфейсу, мережевий стек, робота з базами даних.
   C) Кодування та декодування відео, аудіообробка, аналіз даних.
   D) Машинне навчання, розпізнавання образів, обробка природних мов.

7. У яких галузях використовується RTOS?
   A) У виробництві автомобілів.
   B) У вбудованих системах та медичній техніці.
   C) В авіаційній промисловості та промисловому устаткуванні.
   D) Варіанти A та B.
   E) Варіанти A, B та C.
8. Чи є обмеження для мікроконтролера для можливості встановлення RTOS?
   A) Так, мікроконтролер повинен мати достатньо ресурсів, щоб працювати з RTOS.
   B) Ні, на будь-який мікроконтролер можна встановлювати RTOS.
   C) Обмеження залежить від версії RTOS, а не від мікроконтролера.
   D) Обмеження існують лише для дорогих мікроконтролерів.
9. Що таке "планувальник"?
   A) Програма для розподілу одного ядра процесора на декілька частин.
   B) Частина ОС, для розміщення задач в оперативній пам'яті.
   C) Частина ОС, яка відповідає за розподіл обчислювальних ресурсів і визначення порядку виконання завдань.
   D) Апаратний пристрій для швидкого зберігання даних.
10. Що таке багатозадачність?
    A) Можливість виконання лише одного завдання одночасно.
    B) Можливість виконання кількох завдань одночасно.
    C) Виконання кількох завдань послідовно, але швидко.
    D) Можливість виконання завдань на кількох різних пристроях одночасно.
11. Що таке операційна система?
    A) Програма для створення документів.
    B) Система, що керує ресурсами комп'ютера.
    C) Програма для створення веб-сайтів.
    D) Апаратний пристрій для зберігання даних.
12. Що таке супер-цикл (Super Loop)?
    A) Методологія програмування на Python.
    B) Нелінійний код, що виконується у безкінечному циклі.
    C) Алгоритм обробки даних у базі даних.
    D) Апаратний цикл у пристроях реального часу.
13. Як досягається багатозадачність на одному ядрі?
    A) За допомогою поділу одного ядра на два і більше.
    B) Шляхом перемикання між різними задачами в короткі проміжки часу.
    C) Шляхом збільшення частоти процесора.
    D) Шляхом використання додаткової оперативної пам'яті.
14. Як досягається багатопотоковість в RTOS?
    A) За допомогою одного потоку, що виконує всі завдання.
    B) За допомогою кількох паралельних потоків, керованих планувальником.
    C) За допомогою послідовного виконання завдань одним потоком.
    D) Шляхом використання лише апаратних засобів без ОС.

ТЕМА 8. ОСНОВНІ АСПЕКТИ СИНХРОНІЗАЦІЇ В RTOS

План лекції

- Виділення пам’яті в FreeRTOS
- Черги
- М’ютекси та семафори
- Програмні таймери

Текстова частина лекції

Виділення пам’яті

RTOS використовує різні типи оперативної пам'яті для зберігання даних та керування ресурсами. Серед них:

1) **Статична пам'ять (Static):**
   - розмір пам'яті визначається на етапі компіляції та залишається незмінним протягом виконання програми;
   - зазвичай використовується для глобальних змінних та статичних об'єктів;
   - зручно використовувати для об'єктів з фіксованим розміром, які потрібні протягом усього життєвого циклу програми та не потребують динамічного виділення пам'яті під час виконання.
2) **Купа (Heap)** – для динамічної пам'яті, що виділяється під час виконання програми.
   - динамічно виділяється та звільняється під час виконання програми;
   - розмір та розміщення об'єктів у купі може змінюватись протягом виконання;
   - використовується для об'єктів, розмір яких важко передбачити на етапі компіляції.
3) **Стек (Stack)** – для зберігання локальних змінних та адреси повернення з функцій.
   - використовується для зберігання локальних змінних, керування викликами функцій, передачі параметрів та адрес повернення з функцій;
   - кожному завданню в RTOS призначається власний стек;
   - зазвичай стек росте в напрямку з вищих адрес до нижніх (назустріч купі);
   - визначення розміру стеку може бути використане для передбачення обсягу використаної пам'яті.

При створенні завдання в FreeRTOS, пам’ять для нього виділяється з купи та складається з двох частин: стеку завдання та блоку управління цим завданням.

Стек завдання:

- кожне завдання має свій власний стек;
- розмір стеку може бути переданий як аргумент при створенні завдання;
- стек завдання може бути виділений статично.

Блок керування завданням (Task Control Block, TCB):

- для кожного завдання створюється TCB, який містить інформацію про стан завдання та його контекст;

- TCB містить такі дані, як: стан завдання, пріоритет, вказівник на стек, вказівники на функції, які мають бути викликані при створенні та видаленні завдання.

В FreeRTOS є кілька способів встановлення розміру стеку для кожного завдання, а також інші корисні функції для визначення розміру пам'яті, наприклад:

- **`configMINIMAL_STACK_SIZE`** – один з параметрів, які можна налаштувати в файлі конфігурації (FreeRTOSConfig.h), який визначає мінімальний розмір стеку за замовчуванням для всіх завдань. Наприклад, при створенні нового завдання можна використовувати такий виклик:

```
xTaskCreate(myFunction, "TaskName", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
```

- **`uxTaskGetStackHighWaterMark()`** – функція для вимірювання використаної пам’яті, що дозволяж визначити скільки пам’яті залишилось в стеці завдання після його виконання. Отримані дані можна використовувати для коригування розміру стеку.
- **`xPortGetFreeHeapSize()`** – функція для отримання інформації про кількість вільної пам’яті у купі (Heap). Загалом, її використання допомагає ефективно керувати пам'яттю та вчасно виявляти потенційні проблеми.

Синхронізація

При використанні спільних ресурсів, таких як пам’ять, пристрої вводу-виводу, файлові системи тощо, можуть виникати конфліктні ситуації.

Синхронізація відповідає за координацію доступу до ресурсів та управління конкуренцією між завданнями для запобігання конфліктам та некоректній роботі.

Синхронізація в RTOS необхідна для:

- **Захисту від конкурентного доступу**. У багатозадачних системах кілька завдань можуть намагатися одночасно отримати доступ до спільних ресурсів.
- **Уникнення «перегонів даних»**. Перегони виникають, коли два або більше завдання конкурують за доступ до одного й того ж ресурсу і можуть одночасно змінювати його стан.
- **Забезпечення порядку виконання операцій**. В деяких випадках важливо, щоб операції виконувалися лише в певному порядку або за певних умов.
- **Управління доступом до критичних секцій коду**. Критичні секції коду – це частини програми, де відбувається доступ до спільних ресурсів. Вони потребують особливої уваги при синхронізації для запобігання конфліктам.
- **Забезпечення правильної роботи переривань**. Переривання можуть впливати на виконання завдань в RTOS, тому важливо синхронізувати їхню роботу з основним кодом програми та іншими перериваннями.

Стан перегонів (Race condition)

Стан перегонів (Race condition) виникає в багатозадачних або багатопроцесорних системах, коли результат виконання програми залежить від того, в якому порядку задачі виконують свої операції. Це може призвести до непередбачуваних результатів, помилок у роботі програми або навіть до її збою.

Основні аспекти стану перегонів:

- Дві задачі або більше одночасно намагаються отримати доступ до спільного ресурсу, пам'яті, файлу або пристрою вводу-виводу.
- Доступ до спільного ресурсу не синхронізований за допомогою відповідних механізмів синхронізації (семафори, м'ютекси).
- Порядок виконання операцій непередбачуваний, тобто результати програми можуть змінюватися залежно від того, яка задача першою виконає свою операцію.

Наприклад, якщо дві задачі одночасно збільшують значення змінної на 1, то фактичне значення змінної може бути меншим, ніж очікуване, через конкуренцію між задачами за доступ до цієї змінної.

Щоб уникнути стану перегонів, важливо правильно синхронізувати доступ до спільних ресурсів за допомогою механізмів синхронізації.

FreeRTOS надає декілька інструментів для синхронізації:

- Черги (Queues).
- М'ютекси (Mutexes).
- Семафори (Semaphores).
- Таймери (Timers).
- Події (Event Groups).

Черги (Queues)

Черги дозволяють передавати дані між потоками безпечним та ефективним способом. Порядок використання черги в FreeRTOS може бути наступним:

1) **Оголошення черги**. Для створення черги використовується функція **xQueueCreate**. При цьому вказується максимальна кількість елементів у черзі та їх розмір:

```
xQueueHandle myQueue = xQueueCreate(10, sizeof(int));
```

2) **Відправлення даних в чергу**. Завдання може відправити дані в чергу за допомогою функції **`xQueueSend`**:

```
int dataToSend = 123;
xQueueSend(myQueue, (void*)&dataToSend, xTicksToWait);
```

де **`myQueue`** – черга в яку необхідно помістити дані **`dataToSend`**; **`xTicksToWait`** – максимальний час, протягом якого завдання буде очікувати вільного місця в черзі, якщо вона повна.

3) **Отримання даних з черги**. Інше завдання може отримати дані з черги за допомогою функції **`xQueueReceive`**:

```
int receivedData;
xQueueReceive(myQueue, (void*)&receivedData, xTicksToWait);
```

Ця функція також може очікувати наявність даних у черзі. Замість значення **`xTicksToWait`** можна встановити макрос **`portMAX_DELAY`**.

portMAX_DELAY – це значення, яке використовується для встановлення максимального таймауту очікування в операціях блокування. Операція блокування буде чекати на виконання протягом нескінченного періоду часу, тобто до тих пір, поки не буде викликана відповідна подія. В середовищі FreeRTOS це зазвичай максимальне значення типу **TickType_t**, який визначається конкретно для кожної архітектури (наприклад, 32-бітної чи 64-бітної).

Також можна перевірити наявність даних у черзі за допомогою функції **uxQueueMessagesWaiting**, що дозволяє визначити кількість елементів у черзі, які чекають на обробку:

```
UBaseType_t messagesWaiting = uxQueueMessagesWaiting(myQueue);
```

Детальніше про роботу з чергами в FreeRTOS можна дізнатись з офіційної документації за посиланням [20].

М'ютекси (Mutexes)

М'ютекс (Mutex) – це механізм синхронізації конкурентного доступу до спільних ресурсів різних завдань або потоків програми. У FreeRTOS м'ютекси є одним з інструментів для запобігання перегонам даних та забезпечення атомарності операцій.

Порядок використання м'ютексів у FreeRTOS:

1) **Створення м'ютекса**. Спочатку потрібно оголосити змінну типу **SemaphoreHandle_t**, яка буде використовуватися для зберігання вказівника на створений м'ютекс. Це може бути глобальна або локальна змінна, залежно від того, як широко м'ютекс повинен бути доступний. Після оголошення змінної, можна створити м'ютекс за допомогою функції **xSemaphoreCreateMutex**, яка повертає вказівник на створений м'ютекс:

   ```
   SemaphoreHandle_t xMutex = xSemaphoreCreateMutex();
   ```

2) **Спроба захоплення м'ютекса.** Завдання, яке бажає отримати доступ до критичного ресурсу, викликає функцію **xSemaphoreTake**:

   ```
   xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);
   ```

   де **xSemaphore** – вказівник на м'ютекс, який потрібно захопити; **xTicksToWait** – параметр, який вказує максимальний час очікування на захоплення м'ютекса (в тактових періодах, ticks). Якщо **xTicksToWait** встановлено на **portMAX_DELAY**, функція буде чекати на захоплення м'ютекса нескінченно довго.

   Під час спроби захоплення м'ютекса виконуються наступні дії:

   - **Перевірка доступності м'ютекса**. Якщо м'ютекс вільний (не захоплений іншим завданням), то процес захоплення вважається успішним, і завдання отримує доступ до критичного ресурсу.
   - **Блокування потоку**. Якщо м'ютекс зараз захоплений іншим завданням, поточне завдання, що намагається його захопити, може бути заблоковано. Це означає, що потік виконання цього завдання призупиняється, поки м'ютекс не звільниться.
   - **Таймаут захоплення**. Якщо час **xTicksToWait** вичерпаний, а м'ютекс не став доступним, то функція повертає значення **pdFALSE**, що вказує на неуспішність операції.

- **Успішне захоплення**. Якщо захоплення м'ютекса пройшло успішно, завдання отримує доступ до критичного ресурсу і може виконувати потрібні операції. Після успішного захоплення м'ютекса функція **`xSemaphoreTake`** повертає значення **`pdTRUE`**.

3) **Звільнення м'ютекса.** Коли завдання закінчує роботу з критичним ресурсом, йому потрібно звільнити м'ютекс, щоб інші завдання могли отримати доступ до ресурсу, це робиться за допомогою функції **`xSemaphoreGive`**:

```
xSemaphoreGive(SemaphoreHandle_t xSemaphore);
```

Детальніше про роботу з м’ютексами в FreeRTOS у офіційній документації за посиланням [21].

Семафори (Semaphores)

М'ютекс є підтипом семафора, використання різних типів семафорів дозволяє програмістам вибирати оптимальний механізм синхронізації в залежності від конкретних потреб програми.

В FreeRTOS існує кілька різновидів семафорів, кожен з яких має свої особливості та використання:

1) **Бінарний семафор:**
   - Має два стани: вільний і захоплений.
   - Коли семафор захоплений, інші завдання, які намагаються захопити його, будуть заблоковані, поки він не буде звільнений.
   - Створюється за допомогою функції **`xSemaphoreCreateBinary`**.
2) **Лічильний семафор:**
   - Може мати будь-яку кількість станів, від 0 до заданого значення.
   - Використовується для контролю над кількістю одночасних доступів до обмеженого ресурсу.
   - Після кожного захоплення семафора його лічильник зменшується, а після звільнення – збільшується.
   - Створюється за допомогою функції **`xSemaphoreCreateCounting`**.
3) **М'ютекс**:
   - Забезпечує ексклюзивний доступ до критичних ресурсів тільки для одного потоку в один момент часу.
   - Ідеально підходить для захисту критичних секцій коду, які повинні виконуватися тільки одним потоком одночасно.
   - Створюється за допомогою функції **`xSemaphoreCreateMutex`**.
4) **Рекурсивний семафор:**
   - Дозволяє тому ж завданню захоплювати семафор більше одного разу без блокування. Використовується для реалізації сценаріїв програмування, де функція може викликатися рекурсивно.
   - Коли завдання виходить з критичної секції, воно повинно відпустити семафор стільки разів, скільки воно його захопило, перш ніж інше завдання зможе його захопити.

- Допомагає уникнути проблеми, відомої як «deadlock» або «взаємне блокування», коли завдання чекає на ресурс, яким воно вже володіє.
- Створюється за допомогою функції **`xSemaphoreCreateRecursiveMutex`**.

Порядок використання лічильного семафору в FreeRTOS:

1) **Оголошення та створення лічильного семафора** з використанням функції **`xSemaphoreCreateCounting`**. Перший аргумент вказує на максимальне значення лічильного семафора (максимальна кількість захоплень), а другий аргумент – на початкове значення лічильного семафора:

```
SemaphoreHandle_t xCountingSemaphore = xSemaphoreCreateCounting(3, 3);
```

2) **Створення функції Task**, яка містить критичну секцію коду та взаємодіє з лічильним семафором:

Примітка. **Критична секція коду** – це місце, де виконується код, який потребує захисту від конкурентного доступу. У цій секції можуть відбуватися операції з спільними ресурсами або змінними, які також доступні з інших задач. **Решта коду задачі** може включати обробку даних, затримку або будь-які інші дії.

```
void Task(void *pvParameters)
{
  while (true)
  {
    // Спроба захоплення лічильного семафора
    if (xSemaphoreTake(xCountingSemaphore, portMAX_DELAY) == pdTRUE)
    {
      // Критична секція коду

      // Звільнення лічильного семафора
      xSemaphoreGive(xCountingSemaphore);

      // Решта коду задачі
    }
  }
}
```

Після виклику **`xSemaphoreTake`**, функція спробує захопити семафор. Якщо семафор доступний (тобто його лічильник більше 0), то виконається його захоплення та функція поверне **`pdTRUE`**. Якщо семафор не доступний (його лічильник дорівнює 0), функція буде очікувати поки семафор не звільниться.

Детальніше про роботу з семафорами в FreeRTOS у офіційній документації за посиланням [21].

Таймери (Timers)

Таймери – це інструмент у програмуванні, що використовується для вимірювання часу, виконання певних дій через вказаний проміжок часу або для планування подій. Вони дозволяють викликати функції у майбутньому або з заданою періодичністю.

Можна виділити два типи таймерів:

- **Апаратні** – це таймери, що зазвичай вбудовані безпосередньо у апаратуру (наприклад, мікроконтролери) та оперують на рівні апаратних лічильників. Вони забезпечують високу точність та ефективність, але їх кількість та характеристики можуть залежати від конкретної апаратури.
- **Програмні (software)** – це таймери, що реалізуються у програмному коді. Вони можуть бути більш гнучкими у використанні та налаштуваннях, але мати меншу точність в порівнянні з апаратними. Кількість програмних таймерів в основному обмежена кількістю доступної оперативної пам'яті.

Порядок використання таймерів в FreeRTOS може бути наступним:

1) **Створення таймера**. Спочатку потрібно створити програмний таймер за допомогою функції **`xTimerCreate`**:

```
TimerHandle_t xTimer;
xTimer = xTimerCreate(name, period, autoReload, timerID, callbackFunction);
```

де **`name`** – ім'я таймера, рядок, що ідентифікує таймер у системі та використовується для відлагодження (тип даних **`const char *`**);

`period` – період або інтервал, що вказує час, через який таймер має спрацювати (тип даних **`TickType_t`**);

`autoReload` – параметр, що вказує, чи таймер має автоматично перезапускатися після кожного спрацювання. Якщо цей параметр встановлено в **`pdTRUE`**, таймер буде автоматично перезапущений, якщо в **`pdFALSE`** – буде спрацьовувати лише один раз.

`timerID` – ідентифікатор таймера, це значення, яке може бути використане для ідентифікації таймера у функціях зворотного виклику або обробників подій (тип даних **`void *`**);

`callbackFunction` – функція зворотного виклику, яка викликається, коли таймер спрацює. Вона виконується у контексті завдання, що створило таймер.

2) **Запуск таймера**. Почати відлік можна за допомогою функції **`xTimerStart`**:

```
xTimerStart(xTimer, xBlockTime);
```

де **`xTimer`** – це дескриптор програмного таймера, що вказує на таймер, який потрібно запустити або перезапустити; **`xBlockTime`** – це час, протягом якого завдання, що викликає функцію, має утримуватись в стані блокування, для забезпечення успішного надсилання команди запуску до черги команд таймера (може бути 0).

Приклад використання таймера в FreeRTOS:

```
// Оголошення змінних для таймера та інших параметрів
TimerHandle_t xTimer;
const TickType_t period = pdMS_TO_TICKS(1000);

// Функція зворотного виклику для таймера
void callbackFunction(TimerHandle_t xTimer)
{
    // Код, який виконується при спрацюванні таймера
    Serial.println("Hello");
}

void setup()
{
    Serial.begin(115200);
    // Створення та налаштування таймера
    xTimer = xTimerCreate("Timer", period, pdTRUE, 0, callbackFunction);
    // Запуск таймера
    if (xTimerStart(xTimer, 0) != pdPASS)
    {
        // Обробка помилки
    }
}
void loop() {}
```

У цьому прикладі таймер налаштований на періодичне викликання функції **`callbackFunction`** кожну секунду та вивід повідомлення **«Hello»**.

Детальніше про програмні таймери в FreeRTOS можна дізнатись з офіційної документації за посиланням [22].

Запитання для самоперевірки

1. На які області розподіляється пам'ять з довільним доступом (RAM)?
2. Що зберігається в статичній області пам'яті (Static)?
3. Що зберігається в купі (Heap)?
4. Що зберігається в стеку (Stack)?
5. З якої частини виділяється пам'ять для нового завдання?
6. На які частини розділяється пам'ять виділена завданню?
7. При заповненні стеку, в якому напрямку він "росте"?
8. Чи можна завданню змінити розмір стеку?
9. Чи можна дізнатись скільки байтів залишилось в стеці завдання?
10. Чи можна визначити кількість вільної пам'яті в "купі" (Heap)?
11. Для чого доцільно використовувати черги (Queue) в RTOS?
12. Що таке FIFO (принцип черги)?
13. Що таке м'ютекс (Mutex)?
14. Що таке семафор (Semaphore)?
15. Для чого використовуються software-таймери в FreeRTOS?

Тести

1) Для чого використовуються software-таймери в FreeRTOS?
   A) Для відліку часу простою системи.
   B) Для автоматичного вимкнення системи після певного періоду неактивності.
   C) Для синхронізації роботи між різними пристроями.
   D) Для виконання певних дій або запуску функцій у певний момент часу.
2) Для чого доцільно використовувати черги (Queue) в RTOS?
   A) Для зберігання великої кількості даних.
   B) Для передачі даних між завданнями безпосередньо та асинхронно.
   C) Для зберігання константних значень.
   D) Черги в RTOS не є ефективним інструментом.
3) З якої частини виділяється пам'ять для нового завдання?
   A) Зі стеку.
   B) З купи.
   C) З глобальної пам'яті.
   D) Зі статичної пам'яті.
4) На які області розподіляється пам'ять з довільним доступом (RAM)?
   A) Код програми та стек.
   B) Стек, черга та купа.
   C) Глобальні дані (статична купа) та стек.
   D) Стек, купа та статичні дані.
5) На які частини розділяється пам'ять виділена завданню?
   A) Текст та дані.
   B) Управлінська інформація та стек завдання.
   C) Стек та купа.
   D) Глобальні та локальні змінні завдання.
6) При заповненні стеку, в якому напрямку він "росте"?
   A) Ліворуч.
   B) В напрямку купи.
   C) Стек не може бути заповненим.
   D) Всі відповіді вірні.
7) Чи можна визначити кількість вільної пам'яті в "купі" (Heap)?
   A) Так, можна отримати інформацію про вільний простір в купі (Heap).
   B) Ні, інформація про купу (Heap) не доступна.
   C) Так, але тільки якщо вона встановлена під час конфігурації RTOS.
   D) Це можливо тільки за допомогою додаткових інструментів.
8) Чи можна дізнатись скільки байтів залишилось в стеці завдання?
   A) Так, можна отримати інформацію про розмір стеку, який залишився вільним.
   B) Ні, інформація про розмір стеку не доступна.
   C) Так, але тільки для головного завдання.
   D) Це залежить від налаштувань компілятора.

9) Чи можна завданню встановити розмір стеку?
   A) Ні, стек завжди має фіксований розмір.
   B) Так, розмір стеку можна встановити при створенні завдання.
   C) Так, але тільки для стеку головного завдання.
   D) Це залежить від конфігурації RTOS.
10) Що зберігається в купі (Heap)?
   A) Глобальні змінні.
   B) Динамічно виділені дані в ході виконання програми.
   C) Локальні змінні функцій.
   D) Код програми.
11) Що зберігається в статичній області пам'яті (Static)?
   A) Динамічно виділені дані.
   B) Дані, що змінюються в ході виконання програми.
   C) Змінні та константи, що не змінюються в ході виконання програми.
   D) Код програми.
12) Що зберігається в стеку (Stack)?
   A) Локальні змінні функцій та адреси повернення.
   B) Глобальні змінні.
   C) Динамічно виділені дані.
   D) Змінні та константи, що не змінюються в ході виконання програми.
13) Що таке FIFO (принцип черги)?
   A) Перший прийшов, перший вийшов.
   B) Перший прийшов, останній вийшов.
   C) Перший вийшов, перший прийшов.
   D) Останній вийшов, останній прийшов.
14) Що таке м'ютекс (Mutex)?
   A) Технологія шифрування даних.
   B) Механізм для синхронізації доступу до ресурсів між різними завданнями.
   C) Тип даних в мові програмування.
   D) Криптографічний пристрій для генерації випадкових чисел.
15) Що таке семафор (Semaphore)?
   A) Пристрій для вимірювання часу в багатозадачній системі.
   B) Об'єкт для управління доступом до ресурсів в багатозадачній системі.
   C) Тип даних в мові програмування.
   D) Об'єкт для перемикання між процесами на ядрі.

ПЕРЕЛІК ПОСИЛАНЬ

1. STM32 32-bit Arm Cortex MCUs. *STMicroelectronics*. URL: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html (дата звернення: 27.02.2024).
2. STM32 Cortex®-M0+ MCUs programming manual. *STMicroelectronics*. URL: https://www.st.com/resource/en/programming_manual/pm0223-stm32-cortexm0-mcus-programming-manual-stmicroelectronics.pdf (дата звернення: 27.02.2024).
3. SoftwareSerial Library. *Arduino*. URL: https://docs.arduino.cc/learn/built-in-libraries software-serial (дата звернення: 27.02.2024).
4. Serial. *Arduino*. URL: https://www.arduino.cc/reference/en/language/functions/communication/serial (дата звернення: 27.02.2024).
5. RS-232 Glossary and Selection Guide. *Texas Instruments*. URL: https://www.ti.com/lit/an/slla607/slla607.pdf (дата звернення: 27.02.2024).
6. SPI. *Arduino*. URL: https://www.arduino.cc/reference/en/language/functions/communication/spi (дата звернення: 27.02.2024).
7. Wire. *Arduino*. URL: https://www.arduino.cc/reference/en/language/functions/communication/wire (дата звернення: 27.02.2024).
8. Ethernet - Ethernet.begin(). *Arduino*. URL: https://www.arduino.cc/reference/en/libraries/ethernet/ethernet.begin (дата звернення: 27.02.2024).
9. Ethernet Shield Web Client. *Arduino*. URL: https://docs.arduino.cc/tutorials/ethernet-shield-rev2/web-client (дата звернення: 27.02.2024).
10. WiFi. *Arduino*. URL: https://www.arduino.cc/reference/en/libraries/wifi (дата звернення: 27.02.2024).
11. ArduinoJson: Efficient JSON serialization for embedded C++. *ArduinoJson*. URL: https://arduinojson.org (дата звернення: 27.02.2024).
12. PubSubClient. *Arduino*. URL: https://www.arduino.cc/reference/en/libraries/pubsubclient (дата звернення: 27.02.2024).
13. Free Public MQTT Broker. *HiveMQ*. URL: https://www.hivemq.com/mqtt/public-mqtt-broker (дата звернення: 27.02.2024).
14. MQTT Websocket client. *HiveMQ*. URL: https://www.hivemq.com/demos/websocket-client (дата звернення: 27.02.2024).
15. AWS IoT | Industrial, Consumer, Commercial, Automotive | Amazon Web Services. *Amazon Web Services, Inc*. URL: https://aws.amazon.com/iot (дата звернення: 27.02.2024).
16. Azure IoT Products and Services. *Microsoft Azure*. URL: https://azure.microsoft.com/en-us/products/category/iot (дата звернення: 27.02.2024).
17. FreeRTOS task states and state transitions described. *FreeRTOS*. URL: https://www.freertos.org/RTOS-task-states.html (дата звернення: 27.02.2024).
18. Comparison of real-time operating systems. *Wikipedia*. URL: https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems (дата звернення: 27.02.2024).

19. This page describes the RTOS xTaskCreate() FreeRTOS API function which is part of the RTOS task control API. *FreeRTOS*. URL: https://www.freertos.org/a00125.html (дата звернення: 27.02.2024).
20. FreeRTOS queue API functions, including source code functions to create queues, send messages on queues, receive messages on queues, peek queues, use queues in interrupts. *FreeRTOS*. URL: https://www.freertos.org/a00018.html (дата звернення: 27.02.2024).
21. FreeRTOS semaphore and mutex API functions vSemaphoreCreateBinary, xSemaphoreCreateCounting, xSemaphoreCreateMutex, xSemaphoreCreateRecursiveMutex, xSemaphoreTake, xSemaphoreTakeRecursive, xSemaphoreGive, xSemaphoreGiveRecursive, xSemaphoreGiveFromISR. *FreeRTOS*. URL: https://www.freertos.org/a00113.html (дата звернення: 27.02.2024).
22. This page lists the FreeRTOS software time API functions, including source code functions to create timers, start timers, delete timers, reset timers, etc. *FreeRTOS*. URL: https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html (дата звернення: 27.02.2024).