EMBEDDED

л.9
Особливості реального світу (2)

Палій Святослав

Український Католицький Університет

APPLIED SCIENCES FACULTY

Переривання

Переривання — це механізм, який дозволяє процесору тимчасово зупинити виконання поточного коду, щоб виконати інший, терміновий код, який відповідає на певну подію або сигнал. Основні концепції використання переривань:

1. **Обробники переривань (Interrupt Handlers, ISR Interrupt Service Routines):**
   - Це спеціальні функції, які виконуються у відповідь на переривання.
   - Їх реалізують так, щоб виконання було швидким і ефективним.
2. **Типи:**
   - *Зовнішні апаратні переривання*: виникають у результаті сигналів від апаратних пристроїв, наприклад, клавіатури або таймера.
   - *Внутрішні апаратні переривання:* виникають в результаті вийняткових ситуацій.
   - *Програмні переривання*: ініціюються самим програмним забезпеченням за допомогою спеціальних інструкцій або викликів.
3. **Реєстрація обробника та налаштування переривань:**
   - Потрібно зареєструвати обробник переривань та пов’язати його з конкретним перериванням, потрібно дозволити переривання
   - У мові С це зазвичай робиться через функції, специфічні для апаратного забезпечення або операційної системи.
4. **Пріоритетність переривань:**
   - Якщо відразу виникає кілька переривань, пріоритети визначають, яке з них має бути оброблене спочатку.
5. **Контекстне збереження:**
   - Перед виконанням обробника переривань процесор зберігає контекст (тобто стан регістрів та включно з вказівником поточного виконання – program counter), щоб після завершення переривання повернутися на те саме місце де перервалася основна програма.
6. **Використання спеціальних бібліотек та функцій:**
   - Зазвичай існують специфічні бібліотеки для роботи з перериваннями, які полегшують їх інтеграцію в код.

Приклади для AVR

Arduino library

```
void setup()
{
  attachInterrupt(digitalPinToInterrupt(2), buttonIsr, FALLING);
}

void buttonIsr(int segment)
{
   // тут обробляємо подію або ставимо прапорець
}
```

AVR Native library

```
void setup()
{
    TIMSK1 |= (1 << OCIE1A); // Включити переривання від таймера
    sei(); // глобальний дозвіл переривань
}

ISR(TIMER1_COMPA_vect) {
    // тут обробляємо подію або ставимо прапорець
}
```

Обробник переривання повинен бути в розумних межах короткий та ефективний. Обробники переривань зазвичай пишуться таким чином, щоб час їх обробки був якомога меншим, оскільки під час їх роботи можуть не оброблятися інші переривання, а якщо їх буде багато (особливо від одного джерела), то вони можуть губитися.

Критерії для вибору обробляти подію відразу в обробнику чи використовувати прапорець і обробляти в головному циклі:

- Даний тип події потребує негайного оброблення
- Даний тип сигналізує про нетривалу подію, яку слід не пропустити
- Переривання використовується для того щоб відреагувати на подію коли процесор мікроконтролера перебуває в режимі сну

В багатьох архітектурах неможливо поставити точку зупинки для відлагодження в обробнику переривання

Типові помилки

Відсутній дозвіл переривання

- Не увімкнути глобальні переривання (наприклад **sei()** для AVR). Це може призвести до того, що жодне переривання не буде оброблене.
- Не дозволити переривання від конкретного джерела переривання

Довготривале виконання обробників переривань:

- Обробник переривань (ISR) повинен бути максимально коротким, оскільки він блокує виконання іншого коду. Довгий та/або неефективний ISR може викликати затримки та пропуск інших критичних подій.
- Окремим випадком є занадто багато переривань, так, що ISR не дають можливості виконуватися головному циклу

Збереження/відновлення контексту:

- Якщо ISR не зберігає значення регістрів (у деяких архітектурах це треба робити вручну) або використовуються ресурси котрі не зберігаються автоматично без їх збереження та відновлення в коді обробника, основна програма буде працювати некоректно після завершення обробки.

Конкуренція за доступ до змінних:

- Обробники переривань і основна програма можуть одночасно використовувати одні й ті самі глобальні змінні.
- Змінні котрі спільно використовуються обробником повинні маркуватися **volatile**
- Складні структури даних можуть призводити до "гонок" даних. Для уникнення цього варто використовувати атомарні операції або блокування переривань (критичні секції) під час доступу до змінної.

Конкуренція за доступ до змінних

```
void setup()
{
    TIMSK1 |= (1 << OCIE1A); // Включити переривання від таймера
    sei(); // глобальний дозвіл переривань
}

ISR(TIMER1_COMPA_vect)
{
    static uint8_t statVarISR;
    performSomeFunction();
}

void performSomeFunction()
{
    static uint8_t statVar;
    // далі код який робить щось корисне
}

void loop()
{
    // …
    performSomeFunction();
    // …
}
```

Тут все добре, оскільки змінна використовується лише в обробнику, котрий не може сам себе перервати

Тут є проблема, оскільки функція performSomeFunction може бути перервана обробником TIMER1_COMPA_vect і таким чином значення statVar може бути змінено «другим викликом» performSomeFunction без відома першого.

Складні структури даних

```
#define MAX_ELEMENTS 100 // Максимальна кількість елементів

typedef struct {
    uint8_t count;                  // Кількість елементів
    uint8_t elements[MAX_ELEMENTS]; // Масив для зберігання елементів
} ElementContainer;

volatile ElementContainer container;

// Функція для додавання елемента
void addElement(ElementContainer *container, uint8_t newElement)
{
    if (container->count < MAX_ELEMENTS)
    {
        container->count++;
        container->elements[container->count] = newElement;
    }
}

// Обробник переривання
ISR(TIMER1_COMPA_vect)
{
    if (container.count > 0)
    {
        sendToUART(container.elements[container.count-1]);
        container.count--;
    }
}

// Основна цикл
void loop(void) {
    // ...
    addElement(&container, 10);
    // ...
}
```

В даній структурі дані мають зміст лише цілісно, кількість повинна завжди співпадати з власне заповненими елементами даних.

Уявіть що переривання відбулося ось в цьому місці
Цілісність даних буде порушення. Кількість елементів вже збільшена а самі дані ще не записані

Цей код передасть в комунікацію байт з невизначеними даними (сміттям) оскільки корисні дані ще не були записані.
І на додачу при поверненні в основний код дані будуть записані на не своє місце.

Складні структури даних (вирішення)

```
#define MAX_ELEMENTS 100 // Максимальна кількість елементів

typedef struct {
    uint8_t count;                  // Кількість елементів
    uint8_t elements[MAX_ELEMENTS]; // Масив для зберігання елементів
} ElementContainer;

volatile ElementContainer container;

// Функція для додавання елемента
void addElement(ElementContainer *container, uint8_t newElement)
{
    if (container->count < MAX_ELEMENTS)
    {
        cli();
        container->count++;
        container->elements[container->count] = newElement;
        sei();
    }
}

// Обробник переривання
ISR(TIMER1_COMPA_vect)
{
    if (container.count > 0)
    {
        sendToUART(container.elements[container.count-1]);
        container.count--;
    }
}

// Основна цикл
void loop(void) {
    // ...
    addElement(&container, 10);
    // ...
}
```

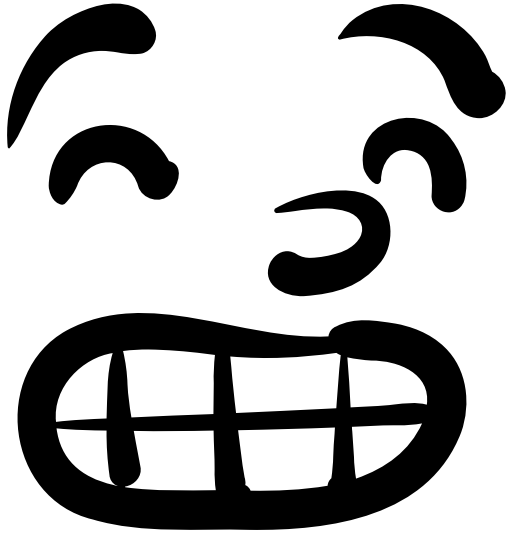
Так ми сформували критичну секцію, котра забезпечує атомарність даних за допомогою заборони і дозволу переривання.

Зауважте що так можна робити тільки у випадку якщо в системі завжди дозволені переривання. Якщо на момент виклику даної функції переривання можуть бути заборонені, то недолік такого методу полягає в тому що ми можемо їх невчасно дозволити.

Багатобайтні дані - це також складні дані

ЗА УМОВИ ЩО ЇХ РОЗРЯДНІСТЬ ПЕРЕВИЩУЄ РОЗРЯДНІСТЬ МІКРОПРОЦЕСОРА

```
void updateData(int32_t newValue)
{
    cli(); // Вимкнути глобальні переривання
    counter = newValue;
    sei(); // Увімкнути глобальні переривання
}
```



Таймер+Переривання

Періодичні переривання від таймера зазвичай використовуються для формування часової бази виконання програми.

Зауважте що популярні функції **delay()** та **millis()** в Arduino (та еквівалентні їх функції в інших середовищах та бібліотеках) використовують переривання від таймера для функціонування. З тих причин їх використання обмежено або неможливо в обробниках переривання.

Тобто програмування з використанням використання **delay()** та **millis()** це також програмування з використанням періодичних переривань від таймера.

Приклад (PSOC) scheduler

```
extern struct BIFlag BurnInFlag;

volatile uint16_t MSCounter=0;
volatile uint8_t SecCounter=0;
volatile uint8_t MinCounter=0;
volatile uint16_t HourCounter=0;

// Minutes from the power on
uint16_t MinOverall=0;

void SchedulerInit(void) {
    Cy_SysTick_Init (CY_SYSTICK_CLOCK_SOURCE_CLK_CPU, 24000u); /* Initialize SysTick with interval = 1ms, using SysClk = 48 MHz as a source */
    Cy_SysTick_SetCallback (0u, SchedulerCallback);     /* Register one of the SysTick callback */
}

void SchedulerCallback(void) {
    MSCounter++;

    if(MSCounter >= 1000) {    // every second
        MSCounter = 0;
        SecCounter++;
        BurnInFlag.BIUartSend = 1;   // send data by UART
        BurnInFlag.BIGetTemperature = 1;  // measure temperature
    }

    if(SecCounter >= 60) {    // every minute
        SecCounter=0;
        MinCounter++;
        MinOverall++;
        BurnInFlag.BIHiTempCheck = 1;  // perform temperature check
    }

    if(MinCounter >= 60) {    // every hour
        MinCounter=0;
        HourCounter++;
    }

    if(0 == MSCounter || 500 == MSCounter)  {
        BurnInFlag.BIMonitoringPin = 1;  // Toggle pins 500ms On, 500ms Off
    }

    if(0 == MSCounter) {    //0ms Bist enabled for SRC port
        BurnInFlag.BistOnP0 = 1; // Bist enabled for SRC port
        BurnInFlag.DSCHP1On = 1; // Discharge on P1 5% duty cycle with period 1sec
        BurnInFlag.BINGDOOn = 1; // Set NGDO on - toggle with 50% duty
    }

    if(50 == MSCounter) { //50 ms
        BurnInFlag.DSCHP1Off = 1; // Discharge off (P1 5% duty cycle with period 1sec = 50msec)
    }
}
```

Приклад (PSOC) task handler

```
void BurnInApp(void)
{
    Cy_USBPD_FETWDT_ClearWatchdog();

    if(BurnInFlag.BIMonitoringPin) {
        BurnInFlag.BIMonitoringPin = 0;
        Cy_GPIO_Inv(PORT1_1_PORT, PORT1_1_PIN);
        Cy_GPIO_Inv(PORT1_5_PORT, PORT1_5_PIN);
    }

    if(BurnInFlag.BIUartSend) {
        BurnInFlag.BIUartSend = 0;
        UARTSendData();

        uint8_t wr_buf[3];
        wr_buf[0] = 0x21u;          /* Write data in index 1 to address in index 0. */
        wr_buf[1] = (uint8_t)((5000u * 1024u) / 1500u);
        wr_buf[2] = (uint8_t)(((5000u * 1024u) / 1500u) >> 8);
        I2C_Write(0x67u, wr_buf, 3);
    }

    if(BurnInFlag.BINGDOOn) {
        BurnInFlag.BINGDOOn = 0;
        NGDO_On();
    }
    if(BurnInFlag.BINGDOOff) {
        BurnInFlag.BINGDOOff = 0;
        NGDO_Off();
    }

    if(BurnInFlag.BistOnP0) {
        BurnInFlag.BistOnP0 = 0;
        BistOnVconnOff(TYPEC_PORT_0_IDX);
    }
    if(BurnInFlag.BistOffP0) {
        BurnInFlag.BistOffP0 = 0;
        BistOffVconnOn(TYPEC_PORT_0_IDX);
    }

    if(BurnInFlag.BIGetTemperature) {
        BurnInFlag.BIGetTemperature = 0;
        temperature = Cy_OTP_GetTemperature();
        // lower code means higher temperature
        highestTemperature = highestTemperature <= temperature ? highestTemperature : temperature;
    }

    if(BurnInFlag.BIRamCheck)
    {
        BurnInFlag.BIRamCheck = 0;
        Cy_GPIO_Inv(PORT1_2_PORT, PORT1_2_PIN);
        RamCheck();
    }
}
```

Приклад (PSOC) main.c

```
********************************************************************************
* Copyright 2023 Cypress Semiconductor
********************************************************************************/
#include "cycfg.h"
#include "BurnIn.h"
#include "scheduler.h"

struct BIFlag BurnInFlag;

void Cy_OnResetUser(void)
{
    CY_SET_REG32(0x40208a5cu, 0x5DB49A55u); // disable OTP
}

uint32_t VconReg = 0;

int main(void)
{
    cybsp_init();     /* Initialize the device and board peripherals */

    BurnInInit();

    __enable_irq();     /* Enable global interrupts */

    SchedulerInit();

    for(;;)
    {
        BurnInApp();
    }
}

void HardFault_Handler(void)
{
    NVIC_SystemReset();
}
```

Декілька задач

```
const int ledPin = 4;

unsigned long previousMillisLed = 0;
unsigned long previousMillisSerial = 0;

const long intervalLed = 500;  // Інтервал для миготіння світлодіода (500 мс)
const long intervalSerial = 2000; // Інтервал для друку в консоль (2000 мс)

void setup() {
    pinMode(ledPin, OUTPUT); // Встановити пін як вихід
    Serial.begin(9600);      // Ініціалізація серійного зв'язку
}

void loop() {
    unsigned long currentMillis = millis();

    // Задача 1: Миготіння світлодіода
    if (currentMillis - previousMillisLed >= intervalLed) {
        previousMillisLed = currentMillis;
        digitalWrite(ledPin, !digitalRead(ledPin)); // Інвертувати стан світлодіода
    }

    // Задача 2: Виведення в серійну консоль
    if (currentMillis - previousMillisSerial >= intervalSerial) {
        previousMillisSerial = currentMillis;
        Serial.println("Привіт, виконую кілька задач одночасно!");
    }
}
```

переваги такого підходу:

- **Простий підхід**
- **Низькі витрати ресурсів на узгодження ресурсів та часу**
- **Строго детермінована послідовність виконання задач**
- Можна додавати нові задачі з різними інтервалами без зміни основної логіки.

Недоліки такого підходу

Відсутність чіткої пріоритетності задач	•Усі задачі виконуються з однаковим рівнем пріоритету, і якщо одна задача займає багато часу, це може вплинути на виконання інших. •Важливі задачі можуть затримуватися через виконання менш важливих задач •За високого навантаження система може не встигати виконати всі задачі вчасно і не має змоги пропускати виконання менш пріоритетних задач.
Відсутність точного керування таймінгом	•Важко забезпечити регулярність подій котрі потребують строгої періодичності (наприклад вивід звуку, чи керування кроковим двигуном, особливо при зміні швидкості руху). Ймовірно зросте джиттер в сигналі котрий буде генеруватися таким чином
Важкість розширення коду	•Додавання нових задач ускладнює логіку програми, особливо якщо інтервали задач взаємозалежні. •Код може стати складним для підтримки та дуже громіздким через збільшення умовних конструкцій
Незахищеність від блокування	•Якщо одна із задач випадково заблокує цикл (наприклад, через нескінченний цикл або інтенсивний підрахунок), усі інші задачі також перестануть виконуватися.

Декілька потоків

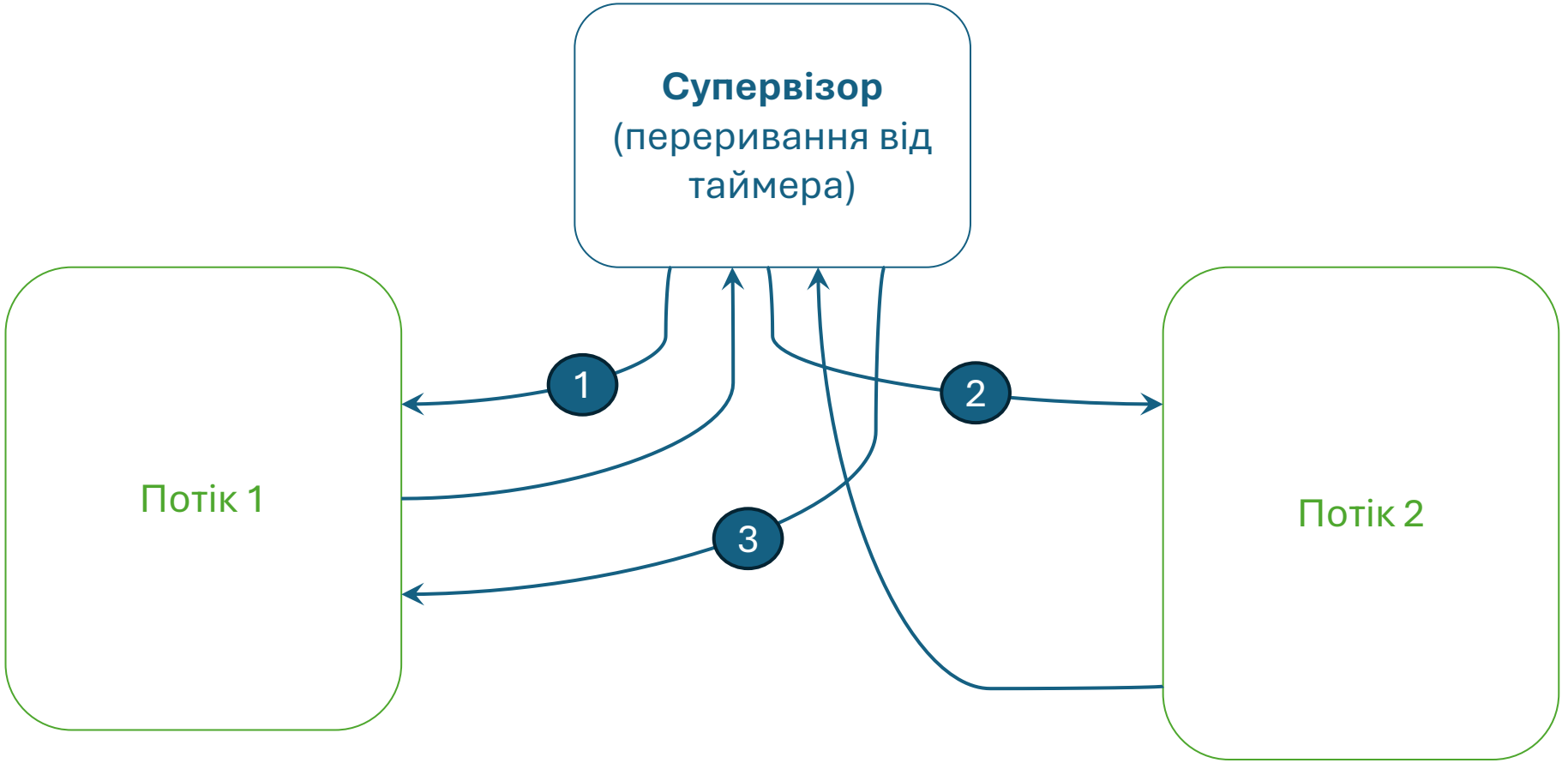
Багатопоточність на одному ядрі є можливою **завдяки *псевдопаралельному виконанню*** потоків. Це означає, що потоки виконуються не одночасно, а по черзі, перемикаючись між ними так швидко, що створюється ілюзія паралельності. Зазвичай таке перемикання здійснюється через планувальник потоків або через кооперативний підхід.

Для реалізації справжньої багатопоточності можна використовувати двоядерні (або й більше) мікроконтролери, такі як ESP32 (2 ядра), які підтримують окремі потоки на кожному ядрі.

Приклад для ESP (Arduino, FreeRTOS)

```
#include <Arduino.h>

void Task1(void *pvParameters) {  // Функція для першого потоку
    while (1) {
        Serial.println("Потік 1: Робота з першою задачею");
        vTaskDelay(1000 / portTICK_PERIOD_MS); // Затримка 1 секунда
    }
}

void Task2(void *pvParameters) {// Функція для другого потоку
    while (1) {
        Serial.println("Потік 2: Робота з другою задачею");
        vTaskDelay(500 / portTICK_PERIOD_MS); // Затримка 500 мілісекунд
    }
}

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

    xTaskCreatePinnedToCore(
        Task1,              // Вказівник на функцію потоку
        "Task 1",           // Ім'я потоку
        1000,               // Розмір стеку для потоку
        NULL,               // Додаткові параметри для потоку
        1,                  // Пріоритет
        NULL,               // Ідентифікатор потоку
        0 );                   // Ядро 0 для виконання потоку

    xTaskCreatePinnedToCore(
        Task2,              // Вказівник на функцію потоку
        "Task 2",           // Ім'я потоку
        1000,               // Розмір стеку для потоку
        NULL,               // Додаткові параметри для потоку
        1,                  // Пріоритет
        NULL,               // Ідентифікатор потоку
        1 );                // Ядро 1 для виконання потоку
}

   void loop() {
    // Основний цикл може залишатися порожнім, оскільки всі задачі виконуються в потоках
}
```

А на одному ядрі ?



Тут слід собі уявити незвичайний обробник переривання, котрий повертає керування не в те місце звідки він був викликаний, а в те місце звідки він був викликаний попереднього разу, і так кожен раз.

Маніпуляція зі стеком

```
#include <avr/io.h>
#include <avr/interrupt.h>

// Функція, в яку ми хочемо перенаправити виконання
void alternateFunction() {
    // Цей код буде виконуватися після переривання
    while (1) {
        digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN)); // Миготіння вбудованого світлодіода
        delay(500); // Затримка
    }
}

ISR(TIMER1_COMPA_vect) {
    uint16_t returnAddress = (uint16_t)alternateFunction; // Нова адреса повернення

    // Маніпуляція стеком
    asm volatile (
        "pop r31\n"          // Забираємо старший байт поточної адреси повернення зі стеку
        "pop r30\n"          // Забираємо молодший байт поточної адреси повернення зі стеку
        "push %B0\n"         // Повертаємо новий старший байт адреси
        "push %A0\n"         // Повертаємо новий молодший байт адреси
        : : "r" (returnAddress) // Передаємо адресу `alternateFunction`
    );
}

void setup() {
    pinMode(LED_BUILTIN, OUTPUT); // Встановлюємо вбудований світлодіод як вихід

    // Налаштування таймера
    TCCR1B |= (1 << WGM12);  // Увімкнення режиму CTC
    OCR1A = 15624;           // Порогове значення для 1 Гц (16 МГц/1024)
    TIMSK1 |= (1 << OCIE1A); // Увімкнення переривання за порівнянням
    TCCR1B |= (1 << CS12) | (1 << CS10); // Переддільник (prescaler) 1024

    sei(); // Увімкнення глобальних переривань
}

void loop() {
    // Основний цикл нічого не робить, усе виконується через переривання
}
```

pop r31: зчитує і вилучає байт зі стеку й записує його у регістр r31.
pop r30: зчитує і вилучає байт зі стеку й записує його у регістр r30.
push %B0: додає старший байт нової адреси повернення (переданої через returnAddress) у стек.
push %A0: додає молодший байт нової адреси повернення (з returnAddress) у стек.
: : "r" (returnAddress): це синтаксис для передачі значення returnAddress із використанням регістрів. Під час виконання це значення розбивається на два байти: молодший (%A0) і старший (%B0).

RTOS

операційна система реального часу, призначена для виконання задач із суворими часовими обмеженнями. У RTOS планувальник забезпечує, щоб задачі виконувалися в чітко визначений час або з мінімальною затримкою. Це робить RTOS ідеальним вибором для вбудованих систем, де важлива передбачуваність поведінки системи.

Основні характеристики:

1. **Планування реального часу:**
   Використовується або/або
   пріоритетне планування (priority scheduling),
   циклічне планування (round-robin scheduling).
   Завдання з вищим пріоритетом завжди отримують доступ до процесорного часу в першу чергу.
2. **Детермінованість:**
   RTOS гарантує, що завдання будуть виконані у фіксовані терміни, що є критично важливим для реальних систем.
3. **Мінімальні затримки:**
   RTOS мінімізує затримки обробки задач і швидко реагує на події (переривання).
4. **Механізми синхронізації:**
   Забезпечуються механізми для координації задач, такі як семафори, м'ютекси, події або черги.
5. **Підтримка багатозадачності:**
   RTOS дозволяє виконувати кілька завдань одночасно, розподіляючи процесорний час між ними.

```
#include <Arduino_FreeRTOS.h>

// Завдання 1: Миготіння світлодіода
void TaskBlink(void *pvParameters) {
    (void) pvParameters;
    pinMode(LED_BUILTIN, OUTPUT);
    while (1) {
        digitalWrite(LED_BUILTIN, HIGH);
        vTaskDelay(1000 / portTICK_PERIOD_MS); // Затримка 1 секунда
        digitalWrite(LED_BUILTIN, LOW);
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}

// Завдання 2: Друк у консоль
void TaskPrint(void *pvParameters) {
    (void) pvParameters;
    while (1) {
        Serial.println("Привіт, RTOS!");
        vTaskDelay(500 / portTICK_PERIOD_MS); // Затримка 500 мс
    }
}

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

    // Створення завдань
    xTaskCreate(TaskBlink, "Blink", 128, NULL, 1, NULL);
    xTaskCreate(TaskPrint, "Print", 128, NULL, 1, NULL);
}

void loop() {
    // Основний цикл порожній, усе виконується в задачах
}
```

FreeRTOS example

xTaskCreate: створення задачі.
vTaskDelay: затримка виконання задачі, яка звільняє процесор для інших задач.

Переваги та недоліки

ПЕРЕВАГИ:

•**Передбачуваність:** Забезпечує гарантоване виконання задач у реальному часі.

• **Масштабованість:** Підходить як для простих, так і для складних проектів. Дає можливість поділити проект на незалежні частини для виконання окремими групами чи особами.

НЕДОЛІКИ:

• **Більше споживання пам'яті** порівняно із простим програмуванням.

• **Використання процесорного часу самою OS.**

• **Складність налаштування,** особливо для новачків. Необхідне вивчення документації та особливостей кокретної OS перед її використанням.

```cpp
#include <Arduino_FreeRTOS.h>
#include <semphr.h>

    SemaphoreHandle_t xSemaphore;

    void Task1(void *pvParameters) {
    while (1) {
        // Захоплення семафора
        if (xSemaphoreTake(xSemaphore, portMAX_DELAY) == pdTRUE) {
            Serial.println("Задача 1 отримала доступ до ресурсу");
            delay(1000); // Емулюємо використання ресурсу
            xSemaphoreGive(xSemaphore); // Звільнення семафора
        }
        vTaskDelay(500 / portTICK_PERIOD_MS);
    }
}

    void Task2(void *pvParameters) {
    while (1) {
        if (xSemaphoreTake(xSemaphore, portMAX_DELAY) == pdTRUE) {
            Serial.println("Задача 2 отримала доступ до ресурсу");
            delay(1000); // Емулюємо використання ресурсу
            xSemaphoreGive(xSemaphore);
        }
        vTaskDelay(500 / portTICK_PERIOD_MS);
    }
}

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

        // Ініціалізація семафора
    xSemaphore = xSemaphoreCreateBinary();
    xSemaphoreGive(xSemaphore); // Дозволяємо доступ до ресурсу

        // Створення задач
    xTaskCreate(Task1, "Task 1", 128, NULL, 1, NULL);
    xTaskCreate(Task2, "Task 2", 128, NULL, 1, NULL);
}

    void loop() {
    // Основний цикл залишається порожнім
}
```

Семафори, мютекси

Типи семафорів:

1.Бінарні семафори (Binary Semaphore):

- Подібний до мютекса (mutex – mutually exclusive)*
  Mutex зазвичай це бінарний семафор з додатковими особливостями, наприклад обмежений одним потоком
- Мають лише два стани: заблокований і дозволений
- Найчастіше використовуються для забезпечення взаємного виключення при доступі до критичних секцій.

2.Лічильні семафори (Counting Semaphore):

- Мають значення, яке вказує кількість "дозволених доступів" до ресурсу.
- Наприклад, якщо значення семафора дорівнює 3, це означає, що до ресурсу можуть одночасно звертатися три задачі.

Як працює семафор:

- **Ініціалізація:**
  Семафор встановлюється на певне початкове значення.
- **Захоплення семафора (Wait або Take):**
  Якщо значення семафора більше 0, потік/задача може отримати доступ до ресурсу, а значення семафора зменшується на 1.Якщо значення дорівнює 0, потік/задача буде заблокований, поки ресурс не стане доступним.
- **Звільнення семафора (Signal або Give):**
  Коли потік/задача завершив доступ до ресурсу, значення семафора збільшується на 1, дозволяючи іншим потокам/задачам отримати доступ.

```ino
#include <Arduino_FreeRTOS.h>
#include <queue.h>

QueueHandle_t xQueue;

void ProducerTask(void *pvParameters) {
    int value = 0;
    while (1) {
        if (xQueueSend(xQueue, &value, portMAX_DELAY) == pdPASS) {
            Serial.print("Значення відправлено: ");
            Serial.println(value);
            value++;
        }
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}

void ConsumerTask(void *pvParameters) {
    int receivedValue;
    while (1) {
        if (xQueueReceive(xQueue, &receivedValue, portMAX_DELAY) == pdPASS) {
            Serial.print("Значення отримано: ");
            Serial.println(receivedValue);
        }
    }
}

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

    // Створення черги для передачі даних
    xQueue = xQueueCreate(5, sizeof(int));
    if (xQueue == NULL) {
        Serial.println("Не вдалося створити чергу");
        while (1);
    }

    // Створення задач
    xTaskCreate(ProducerTask, "Producer", 128, NULL, 1, NULL);
    xTaskCreate(ConsumerTask, "Consumer", 128, NULL, 1, NULL);
}

void loop() {
    // Основний цикл залишається порожнім
}
```

Черги

Основні характеристики черги:

- **Використання в багатозадачності:**
  - черги дозволяють потокам обмінюватися даними або повідомленнями.
- **Розмір черги:**
  - Може бути статичним (фіксована кількість елементів) або динамічним.

Типи черг:

- **Звичайна черга (FIFO - First In, First Out):**
  - Дані додаються в "хвіст" черги (enqueue) і зчитуються з її "голови" (dequeue).
  - Приклад: черга людей у магазині - перша людина в черзі обслуговується першою.
- **Пріоритетна черга:**
  - Кожен елемент має свій пріоритет, і обробляється в порядку пріоритету, а не в порядку надходження.
- **Циркулярна (кільцева) черга:**
  - Кінець черги зв’язується з її початком, щоб ефективно використовувати пам’ять.
- **Блокуюча черга (Blocking Queue):**
  - Використовується в багатопоточних середовищах, де потік очікує, поки черга стане доступною для запису або зчитування.

Приклади RTOS:

FreeRTOS:

- Відкрита та популярна RTOS для вбудованих систем із широкою підтримкою мікроконтролерів, включаючи Arduino та ESP32.
- Легка вага: Підходить для систем із обмеженими ресурсами.
- Підтримка великої кількості мікроконтролерів.
- Зручний API для створення задач, використання семафорів, м'ютексів, черг тощо.
- Інтеграція з бібліотеками IoT від AWS (Amazon FreeRTOS).

Zephyr: (by Linux foundation)

- Інноваційна платформа реального часу для IoT-пристроїв.
- Сучасна архітектура з великою увагою до безпеки.
- Підтримка багатьох платформ, від мікроконтролерів до IoT-рішень.
- Вбудована підтримка підключення до мереж (Wi-Fi, Bluetooth, LoRa).
- Потужна система керування енергоспоживанням.

ThreadX: (Azure RTOS by Microsoft)

- Використовується в комерційних продуктах і забезпечує високу продуктивність.
- Висока продуктивність із дуже швидким перемиканням задач.
- Малий розмір ядра, що підходить для вбудованих систем.
- Інтеграція з Azure IoT для створення хмарних рішень.
- Потужна бібліотека для синхронізації, таймерів, управління пам'яттю.

VxWorks:

- Популярна RTOS у промислових і критичних системах, таких як авіоніка.
- Підтримка жорстких обмежень реального часу.
- Висока безпека та відповідність стандартам (наприклад, авіоніка DO-178C).
- Підтримка багатозадачності, багатопоточності та віртуалізації.
- Підтримка як простих, так і складних процесорів.

Приклади RTOS:

RIOT OS:

- Легка RTOS для IoT з відкритим кодом.
  - Дуже мала вага, підходить для мікроконтролерів із мінімумом ресурсів.
  - Підтримка IPv6, 6LoWPAN для IoT-з'єднань.
  - Ефективне управління енергоспоживанням.
  - Розширюваний модульний підхід.

MBED OS: (by ARM)

- розроблена компанією Arm, спеціально для пристроїв Інтернету речей (IoT).

Щиро дякую!