EMBEDDED

л.14
Безпека та надійність

Палій Святослав

Український Католицький Університет

APPLIED SCIENCES FACULTY

Безпека

Безпека та правильне функціонування важливе не тільки в embedded

Але в embedded є ще додаткові фактори які впливають на правильність, безпечність та надійність системи:

- Помилки програмного коду
- Помилки (ліміти) апаратної частини
- Поломки апаратної частини
- Вихід реальних величин за очікувані межі.
- Завади

Як уникати помилок в embeded рішеннях

1. Планування

- **Чіткі вимоги:** Визначте точні вимоги до програмного забезпечення перед початком розробки.
- **Дизайн системи:** Розробіть архітектуру та структуру коду, яка відповідає вимогам і дозволяє масштабування. ***Чим швидше почати писати код тим довше його прийдеться писати та виправляти.***
- **Запас по надійності:** використовуйте методи котрі дозволяють детектувати помилкові стани (біти парності, CRC). Розглядайте варіанти дублювання елементів системи для збільшення надійності.
- **BIST:** вбудовуйте можливість самотестування сиситеми

2. Дотримуйтесь стандартів

- **Кодування:** Використовуйте стандарти програмування (наприклад, MISRA для C/C++, CERT-C).
- **Перевірки:** Використовуйте інструменти статичного аналізу коду для виявлення потенційних проблем.

3. Автоматизуйте тестування

- **Юніт-тести:** Напишіть тести для перевірки кожної функції окремо.
- **Інтеграційні тести:** Переконайтесь, що всі компоненти працюють разом без збоїв.
- **Тестування corner/edge cases :** Моделюйте виняткові ситуації, щоб перевірити стійкість системи.
- **Регулярні перевірки:** Проводьте ретроспективи для аналізу помилок і покращення процесів.

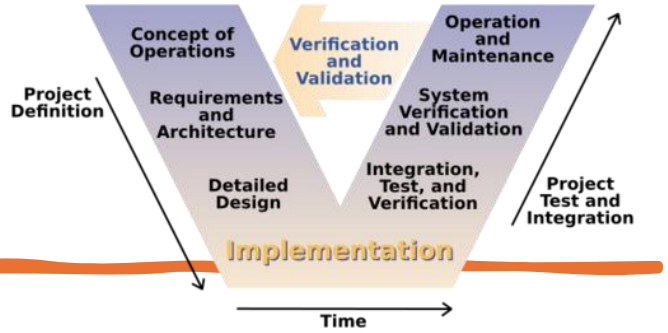
4. Використовуйте контроль версій та code review

- **Код-рев'ю:** Нехай ваш код перевіряють інші члени команди для виявлення можливих проблем**.**
- **Git:** Завжди тримайте резервну копію коду та використовуйте системи контролю версій.

5. Навчайте команду

- **Тренінги:** Навчайте розробників новітніх інструментів і методик.
- **Обмін знаннями:** Проводьте обговорення помилок та уроків, які з них винесли.

V-модель

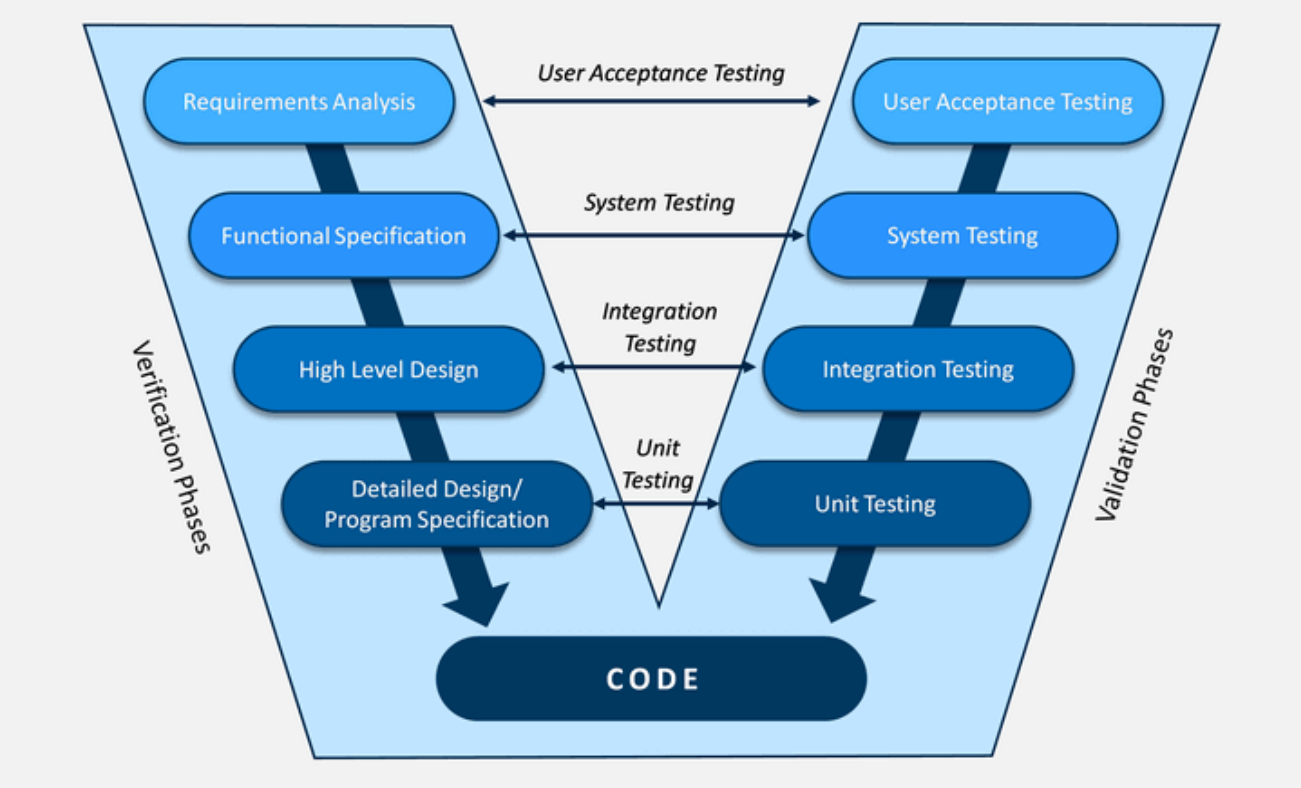


V-модель - це методологія розробки програмного забезпечення, яка підкреслює важливість перевірки на кожному етапі життєвого циклу розробки. Вона є розширенням водоспадної моделі розробки, але з акцентом на тестування.

Ліва сторона "V" представляє етапи розробки (наприклад, аналіз вимог, проєктування системи).

Права сторона "V" відповідає етапам тестування (наприклад, модульне тестування, системне тестування).

Нижня точка "V" — це етап кодування, який з'єднує розробку та тестування.



MISRA-C

Основні риси **MISRA** (Motor Industry Safety and Reliability Association) - це неймовірна увага до деталей та надзвичайна скрупульозність у забезпеченні безпеки.

Автори стандарту не просто зібрали всі можливі «вразливості» мов програмування C та C++, але ретельно опрацювали міжнародні стандарти цих мов і постаралися виписати всі можливі способи помилитися.

А потім вони додали додаткові правила щодо розвитку коду - щоб в уже чистий код було складніше внести нову помилку.

З одного боку, є багато важливих і корисних правил, які варто дотримуватися завжди, незалежно від призначення вашого проекту. Здебільшого вони спрямовані на усунення невизначеної/залежної від реалізації поведінки.

Наприклад:

- **Не використовуйте значення не ініціалізованої змінної.**
- **Усі non-void функції мають повертати значення.**
- **Лічильник циклу не повинен мати тип float.**
- **Не повинні використовуватися вісімкові (octal) константи (наприклад uint16_t a = 018)**
- **Велика увага до приведення типів та їх сумісності.**

Ще правила:

- Не використовуйте goto
- Кожен switch повинен містити default
- Якщо присутнє else if то повинен бути завершальний безумовний else
- Не повинно бути недосяжного коду, або коду котрий не використовується
  - Приклад Ariane-5
- Не повинно бути закоментованих фрагментів коду
- Обмеження адресної арифметики (допустимо [] та *++*)

- Суфікс 'L’ повинен завжди бути заголовною літерою, щоб не спутати ‘l’ з ‘1’ або ‘I’
- Не використовуйте рекурсію (невеликий стек мікроконтролера може переповнитися)
- Тіла операторов *if*, *else*, *for*, *while*, *do*, *switch* завжди повинні бути в { }
- Заборонено динамічне виділення пам’яті (є шанси недачі при виділенні з кучі через фрагментацію пім’яті особливо в мікроконтролерах)

Знайдете помилку ?

```
for (int i = 0; i < n; ++i);
{
    do_something();
}
```

```
if (student_failed_exam = true)
{
    send_to_repeated_exam();
}
else
{
    congratulate_student();
}
```

Знайдете помилку ?

```
for (int i = 0; i < n; ++i);
{
    do_something();
}
```

Цикл тут і закінчився, все решту виконається один раз

У завантаженому коді порушено **MISRA C: 14.2**. Це правило вимагає уникати пустих операторів у циклах for, while або do. У коді наявна крапка з комою (;) одразу після заголовка циклу for, що створює пустий оператор у тілі циклу:

Тут використано присвоєння, отже всі підуть на перездачу

MISRA C: 13.4 Це правило забороняє використовувати результат операції присвоєння в інших виразах.

```
if (student_failed_exam = true)
{
    send_to_repeated_exam();
}
else
{
    congratulate_student();
}
```

Чи все ж можна порушувати?

Правила девіації MISRA дозволяють керувати випадками, коли неможливо або недоцільно дотримуватись певних правил стандарту. Стандарт MISRA передбачає ситуації, у яких відхилення від правил можуть бути виправданими, але вони повинні бути добре задокументовані.

Основні аспекти девіації:

Документування відхилень:

Required-правила: Відхилення можливе, але необхідно надати аргументовану документацію.

Відхилення має включати:

1. Номер порушеного правила.
2. Місце розташування порушення у коді.
3. Обґрунтування, чому порушення є допустимим.
4. Докази, що порушення не впливає на безпеку або функціональність.

Категорії правил:

1. **Mandatory**: Ці правила не можна порушувати.
2. **Required**: Порушення дозволене за умови документування.
3. **Advisory**: Правила, яким не обов’язково слідувати, і відхилення не потребують пояснень.

Статичний аналіз

Статичний аналіз коду — це процес автоматичного перевірки вихідного коду програм для виявлення помилок, вразливостей та невідповідностей без виконання програми.

Як це працює:

1. Статичний аналіз виконується за допомогою спеціальних програм, які перевіряють код на основі правил, стилів та шаблонів.
2. Виявляє потенційні помилки, які можуть виникнути під час виконання програми.

Переваги:

1. **Раннє виявлення проблем:** Помилки та вразливості знаходяться ще до етапу виконання.
2. **Покращення безпеки:** Виявляє вразливості, які можуть бути використані для атак.
3. **Дотримання стандартів:** Перевіряє відповідність коду правилам програмування (наприклад, стандартам MISRA, CERT тощо). Хоча також може перевіряти на правильність без класифікації знайдених помилок до правил MISRA чи CERT.

Обмеження:

1. Не виявляє проблем, які виникають тільки під час виконання програми.
2. Може давати помилкові спрацьовування (false positives), що потребує додаткового часу на аналіз розробником.

Приклади інструментів:

1. **Lint:** Один із перших і найвідоміших інструментів для статичного аналізу коду.
2. **CPPCheck:** Більш сучасний інструмент з відкритим кодом.
3. **PVS Studio:** Доступний безоплатно для студентів
4. **Coverity:** Використовується для виявлення дефектів у промисловому масштабі.

Online Demo - Cppcheck

Чи тут все в порядку?

```c
#include <stdio.h>

   void printMessage() {
    char message[5];
    strcpy(message, "Hello, world!");
    printf("%s\n", message);
}

   int main() {
    printMessage();
    return 0;
}
```

```c
#include <stdio.h>

void calculate(int a, int b) {
    int result;
    if (b != 0) {
        result = a / b;
    }
    printf("Result: %d\n", result);
}

int main() {
    calculate(10, 0);
    return 0;
}
```

```c
#include <stdio.h>

   void printMessage() {
    char message[5];
    strcpy(message, "Hello, world!");
    printf("%s\n", message);
}

   int main() {
    printMessage();
    return 0;
}
```

Переповнення буфера:
Рядок "Hello, world!" перевищує розмір масиву message[5]

```c
#include <stdio.h>

void calculate(int a, int b) {
    int result;
    if (b != 0) {
        result = a / b;
    }
    printf("Result: %d\n", result);
}

int main() {
    calculate(10, 0);
    return 0;
}
```

Use of undefined variable
Якщо b рівне 0 то значення result не визначене

Unit Test & Code Coverage

Це метод тестування, при якому перевіряється окремий "модуль" або компонент коду, наприклад, функція.

Задача: переконатися, що окремий блок коду виконує свою задачу правильно. Тоді збільшуються шанси але не гарантується що система котра складається з «правильних» модулів також працює правильно.

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

void test_add() {
    assert(add(2, 3) == 5); // Очікуваний результат: 5
    assert(add(-1, 1) == 0); // Очікуваний результат: 0
}
```

Легко знайти помилки на ранніх стадіях.
Зменшення ризику виникнення дефектів у майбутніх оновленнях.

```
void f2()
{
    const char *p = NULL;
    for (int i = 0; str[i] != '\0'; i++)
    {
        if (str[i] == ' ')
        {
            p = str + i;
            break;
        }
    }

    // p is NULL if str doesn't have a space. If str always has a
    // a space then the condition (str[i] != '\0') would be redundant
    return p[1];
}
```

Code Coverage (Покриття коду тестами):
Це показник, який показує, наскільки добре ваш код перевірений тестами. Вимірюється як відсоток виконаних інструкцій під час виконання тестів.

покриття тестами стрічок коду (Line Coverage): Перевіряє, чи всі стрічки коду були виконані під час тестування.

Покриття тестами можливих шляхів (Branch Coverage): Перевіряє, чи всі можливі шляхи виконання програми були протестовані.

Інструменти: Наприклад, **gcov**.

Зв'язок між Unit Test і Code Coverage: Unit tests є основою для досягнення високого показника **code coverage**.

Але навіть 100% покриття стрічок коду не гарантує, що весь код є безпомилковим. Воно лише свідчить, що відсутні фрагменти коду, котрі взагалі не були протестовані.

Watchdog

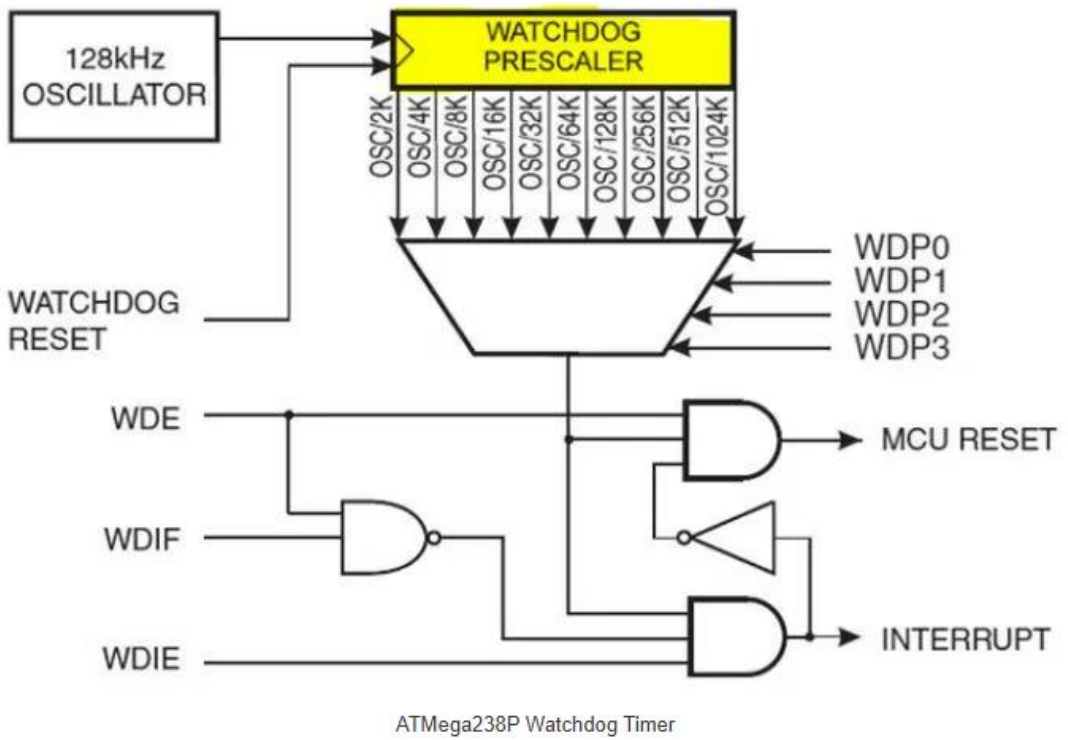
На рівні «ідеї»: є собака (dog) і її потрібно регулярно годувати, якщо собаку вчасно не погодувати вона вкусить.

У програмуванні **watchdog** (таймер сторожового контролю) — це механізм, який використовується для виявлення та відновлення системи у разі її зависання. Основна ідея полягає в тому, що програма або система повинна періодично "перезапускати" таймер, щоб підтвердити, що вона працює нормально. Якщо таймер не перезапускається вчасно, це означає, що система зависла, і таймер виконує заздалегідь визначену дію, наприклад, перезавантаження системи.

Розглянемо на прикладі ATmega328P

Окремий внутрішній осцилятор забезпечує роботу сторожового таймера (watchdog). Таймер перезапуску сторожового таймера може бути налаштований на інтервал від 16 мс до 8 с за допомогою дільника (Prescaler).

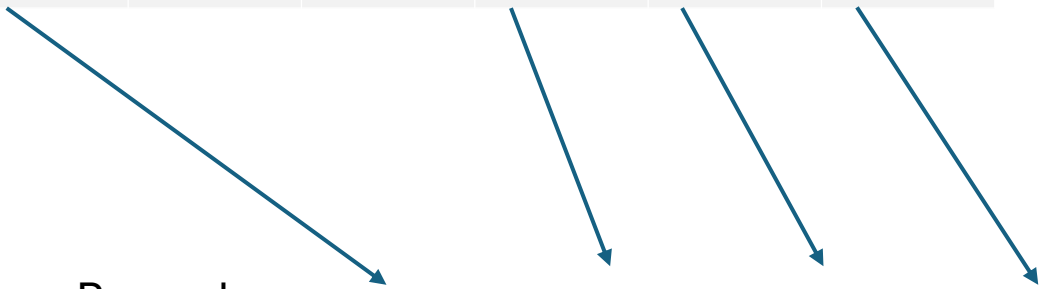
Сторожовий таймер контролюється регістром WDTCSR, як показано нижче. Усі біти, за винятком біта 7 (тільки для читання), можуть бути доступними для читання та запису. Сторожовий таймер з усіма регістрами та контрольними бітами наведено нижче.



ATMega238P Watchdog Timer

Bit	7	6	5	4	3	2	1	0
(0x60)	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0
Read/ Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Bit	7	6	5	4	3	2	1	0
	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0
Access	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial	0	0	0	0	0	0	0	0



Prescaler

WDP3	WDP2	WDP1	WDP0	Oscillator Cycle	Time-out
0	0	0	0	2K	16ms
0	0	0	1	4K	32ms
0	0	1	0	8K	64ms
0	0	1	1	16K	0.125s
0	1	0	0	32K	0.25s
0	1	0	1	64K	0.5s
0	1	1	0	128K	1s
0	1	1	1	256K	2s
1	0	0	0	512K	4s
1	0	0	1	1024K	8s

Bit	7	6	5	4	3	2	1	0
	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0
Access	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial	0	0	0	0	0	0	0	0

Біт **WDCE** (Watchdog Change Enable) у мікроконтролерах ATmega означає **дозвіл на зміну сторожового таймера**. Він використовується для внесення змін до налаштувань сторожового таймера. Щоб виконати зміни, біт WDCE потрібно встановити у "1" разом із бітом WDE (Watchdog Enable). Це розпочинає часову послідовність, протягом якої налаштування сторожового таймера можуть бути змінені (протягом чотирьох тактів після встановлення біта WDCE). Після завершення цієї послідовності біт WDCE автоматично скидається, що унеможливлює подальші зміни.

Fuse **WDTON** в мікроконтролерах ATmega є конфігураційним бітом, який визначає, чи буде сторожовий таймер увімкнений чи вимкнений за замовчуванням після скидання (reset). Коли запобіжник WDTON встановлений в “0”, сторожовий таймер постійно увімкнений, і його роботу неможливо вимкнути під час виконання програми. Це забезпечує механізм для відновлення системи у разі збоїв у програмному забезпеченні або зависання.

Основні моменти щодо WDTON:

- **Призначення:** Примусово забезпечує, активність сторожового таймера, забезпечуючи механізм безпеки для критичних застосувань.
- **Конфігурація:** Запобіжник налаштовується під час програмування та не може бути змінений динамічно з програмного забезпечення.
- **Сфери застосування:** Ідеально підходить для систем, де надійність є надзвичайно важливою, таких як вбудовані системи в медичних пристроях або промисловій автоматизації.

WDTON Fuse	WDE	WDIE	Mode	Action
1	0	0	Stop	No
1	0	1	Interrupt	Interrupt
1	1	0	System Reset	System Reset
1	1	1	Interrupt + System Reset	Interrupt -> System Reset
0	x	x	System Reset	System Reset

Bit	7	6	5	4	3	2	1	0
	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0
Access	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial	0	0	0	0	0	0	0	0

Біт **WDCE** (Watchdog Change Enable) у мікроконтролерах ATmega означає **дозвіл на зміну сторожового таймера**. Він використовується для внесення змін до налаштувань сторожового таймера. Щоб виконати зміни, біт WDCE потрібно встановити у "1" разом із бітом WDE (Watchdog Enable). Це розпочинає часову послідовність, протягом якої налаштування сторожового таймера можуть бути змінені (протягом чотирьох тактів після встановлення біта WDCE). Після завершення цієї послідовності біт WDCE автоматично скидається, що унеможливлює подальші зміни.

Біт **WDIF** (Watchdog interrupt Flag) Вказує що було активоване переривання від watchdog.

Fuse **WDTON** в мікроконтролерах ATmega є конфігураційним бітом, який визначає, чи буде сторожовий таймер увімкнений чи вимкнений за замовчуванням після скидання (reset). Коли запобіжник WDTON встановлений в “0”, сторожовий таймер постійно увімкнений, і його роботу неможливо вимкнути під час виконання програми. Це забезпечує механізм для відновлення системи у разі збоїв у програмному забезпеченні або зависання.

Основні моменти щодо WDTON:

- **Призначення:** Примусово забезпечує, активність сторожового таймера, забезпечуючи механізм безпеки для критичних застосувань.
- **Конфігурація:** Запобіжник налаштовується під час програмування та не може бути змінений динамічно з програмного забезпечення.
- **Сфери застосування:** Ідеально підходить для систем, де надійність є надзвичайно важливою, таких як вбудовані системи в медичних пристроях або промисловій автоматизації.

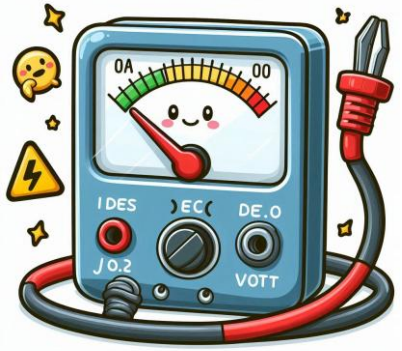
WDTON Fuse	WDE	WDIE	Mode	Action
1	0	0	Stop	No
1	0	1	Interrupt	Interrupt
1	1	0	System Reset	System Reset
1	1	1	Interrupt + System Reset	Interrupt -> System Reset
0	x	x	System Reset	System Reset

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

#define wdt_reset() __asm__ __volatile__("wdr")

const int led = 13;

ISR(WDT_vect) {
  static boolean led_out = HIGH;
  digitalWrite(led, led_out);
  output = !led_out;
}

void setup() {
  Serial.begin(9600);
  pinMode(led, OUTPUT);
  digitalWrite(led, LOW);

  cli();
  wdt_reset();
  WDTCSR |= (1 << WDCE) | (1 << WDE);                 // дозволяємо зміни і вмикаємо WDT
  WDTCSR = (1 << WDIE) | (1 << WDP2) | (1 << WDP1);  // дозволяємо переривання та конфігуруємо прескалер (~1 s)
  sei();
}

void loop() {
  delay(1500);   // при затримках більше 1 сек watchdod буде активуватися
  wdt_reset();
}
```

```
void taskN_handler()
{
    watchdog *wd = watchdog_create(100); /* Створити програмний WDT */

    while (task1_should_run)
    {
        watchdog_feed(wd);
    }
    watchdog_destroy(wd);
}

void watchdog_task_handler()
{
    int i;
    bool feed_flag = true;
    while(1)
    {
        /* Перевірити чи якийсь симульований WDT має timeout */
        for (i = 0; i < getNOfEnabledWatchdogs(); i++)
        {
            if (watchogHasTimeout(i)) {
                    feed_flag = false;
                    break;
            }
         }

         if (feed_flag)
             WatchdogFeedTheHardware();

         task_sleep(10);
}
```

Приклад симульованого WDT в багатозадачній системі

LVD, brownout detector (BOD)

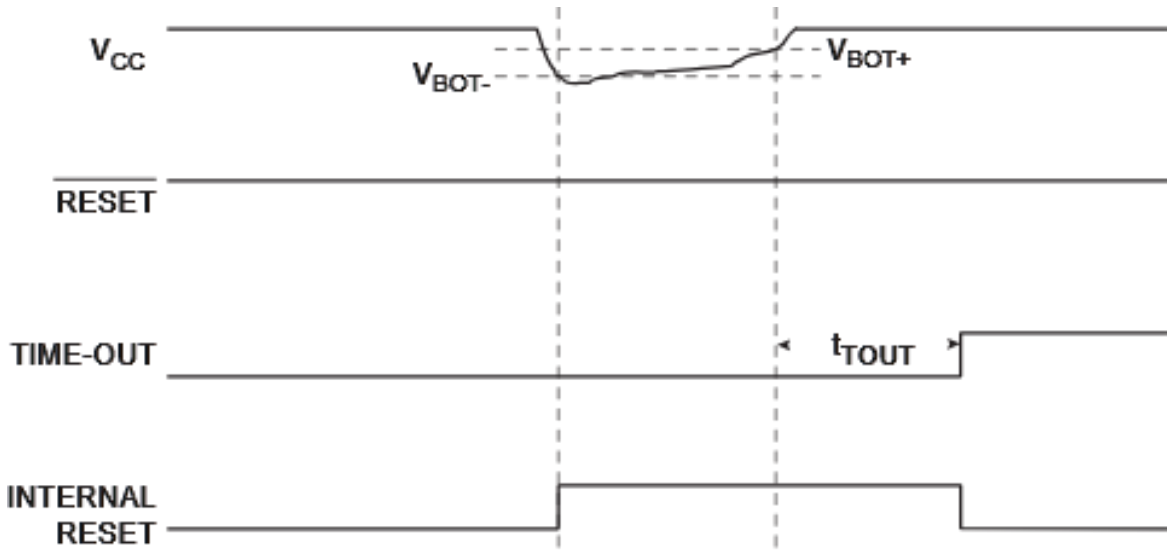


Детектор зниження напруги — це схема або функція в електронних пристроях, включаючи мікроконтролери, яка контролює рівень напруги живлення під час роботи. Його основна задача полягає у виявленні ситуацій, коли напруга падає нижче певного порогу, що може призвести до ненадійної роботи або збою системи.

- Визначає зниження напруги (brownouts), яке виникає, коли напруга живлення тимчасово падає нижче необхідного рівня для стабільної роботи.
- При виявленні зниження напруги детектор зазвичай викликає перезавантаження або перевантаження та утримання системи, щоб уникнути непередбачуваної поведінки.
- Гарантує, що пристрій запускається правильно після повернення напруги до нормального рівня.

ATmega має вбудовану схему виявлення зниження напруги, яка контролює рівень напруги VCC під час роботи, порівнюючи його із заданим пороговим значенням. Порогове значення для BOD може бути вибрано за допомогою fuse **BODLEVEL**: **2,7V** (не запрограмований) або **4,0V** (BODLEVEL запрограмований). Порогове значення має гістерезис, щоб забезпечити виявлення зниження напруги без брязкоту.

Схема BOD може бути включена або вимкнена за допомогою fuse **BODEN**. Якщо BOD увімкнений, і напруга VCC зменшується до значення нижче порогового рівня $\mathbf{V_{BOT-}}$, негайно активується скидання (reset) через зниження напруги. Коли V_{CC} збільшується понад пороговий рівень $\mathbf{V_{BOT+}}$, таймер затримки запускає MCU після завершення періоду очікування $\mathbf{t_{TOUT}}$.



Щиро дякую!