Лекція 10
ПРОГРАМУВАННЯ AVR МОВОЮ С

1. Основні компілятори Сі для МК AVR.

Існує декілька Сі та С++ компіляторів для МК AVR:

- **AVR Toolchain** (**WinAVR**) – безкоштовний Сі компілятор для AVR Studio на основі GCC-компілятора. Створює дуже швидкий і компактний код.
- **CodeVisionAVR** – інтегроване середовище розробки програмного забезпечення для мікроконтролерів сімейства Atmel AVR. Представляє собою компілятор мови С, графічну оболонку і генератор шаблонів програм. Крім стандартних бібліотек мови С, компілятор має бібліотеки для роботи з рідкокристалічними індикаторами, різними шинами, деякими датчиками температури, багатьма модулями пам'яті. Також в CodeVisionAVR є автоматичний генератор шаблонів програм, який дає можливість за короткий час отримати готовий код для роботи багатьох функцій МК. Має вбудований програмний модуль для прошивання і конфігурування МК. Комерційний продукт (150 €).
- **IAR Embedded Workbench for AVR** – інтегроване середовище розроблення та налагодження програм для мікроконтролерів AVR з допомогою мови С, С++ і асемблера. У нього входять компілятор мови С і С++, асемблера, компонувальник і відладчик, при цьому можлива взаємодія із зовнішніми програмами типу AVR Studio. Вимагає складного налаштування, не має прикладів в інсталяції й не має генератора початкового коду. Компілятор IAR генерує швидкий і компактний код. Комерційний продукт (1295-1995 €).
- **ImageCraft C for AVR** – інтегроване середовище розробки з оптимізуючим компілятором, розробленим спеціально для AVR. Для ініціалізації периферії містить модуль-генератор Application Builder. Комерційний продукт (250-550 $).
- **MikroC PRO for AVR** – інтегроване середовище розробки з Сі-компілятором стандарту ANSI, широкий набір бібліотек для апаратних засобів. Комерційний продукт (200-250 $).

Курс лекцій у розділі мови Сі зорієнтований суто на синтаксис та бібліотеки безкоштовного компілятора WINAVR, який зараз перейшов «під крило» Atmel та має назву AVR Toolchain.

2. Типи даних мови Сі компілятора WINAVR.

У табл. 1 представлені числові типи даних, що використовуються у Сі-програмах для компілятора WINAVR. Представлення цілочисельних чисел аналогічне їхньому представленню на асемблері: 0b10101010, 0252, 170, 0xAA.

Таблиця 1. Типи даних мови Сі для компілятора WINAVR

Стандартний	Користу-вацький	К-сть байт	Діапазон значень
char*		1	
signed char	int8_t	1	-128 … 127
unsigned char	uint8_t	1	0…255
short		2	-32 768…32 767
unsigned short		2	0…65535
int	int16_t	2	-32 768…32 767
unsigned int	uint16_ t	2	0…65535
long	int32_t	4	-2 147 483 648…2 247 483 647
unsigned long	uint32_t	4	0 … 4 294 967 295
long long	int64_t	8	-9 223 372 036 854 775 808 … 9 223 372 036 854 775 807
unsigned long long	uint64_t	8	0..18 446 744 073 709 551 615
float		4	$\pm 1,175 \cdot 10^{-38} \dots \pm 3,402 \cdot 10^{38}$
double		8	$\pm 2,2 \cdot 10^{-308} \dots \pm 1,8 \cdot 10^{308}$

* *знаковий чи беззнаковий цей тип даних визначається опцією компілятора.*

.3. Бітова арифметика.

а. Бітові поля. Програмуючи мовою асемблер, ми часто використовували окремі байти як резервуари для 8-ми прапорців (бітів), за допомогою яких ми реалізовували логіку програми. Такий підхід давав нам можливість економити регістри загального призначення. Мовою Сі ми також можемо реалізувати такий підхід, працюючи з окремими бітами вибраного байта. Для цього ми оголошуємо певну змінну типу unsigned char та за допомогою порозрядних операцій працюємо з її окремими бітами.

Інший підхід полягає в оголошені структури з бітовими полями.

```
struct Flags
{
        unsigned f0:1;
        unsigned f1:1;
        unsigned f2:1;
        unsigned f34:2;
        unsigned f57:3;
};
```

Доступ до окремих бітів байта виконується таким чином:

```
struct Flags  cond;
cond.f0=1;
cond.f1=0;
cond.f2=0;
cond.f34=3;          // 0b11
cond.f57=5;          // 0b101
```

Змінна cond розміщується компілятором в оперативній пам'яті SRAM. Тому зміна значень окремих бітів виконується за допомогою порозрядних операцій та числових масок, які обчислюються компілятором.

<u>Згенерований дисасемблером код програми</u>

```
17:      cond.f0=1;
+0000003B:   8189    LDD    R24,Y+1      Load indirect with displacement
+0000003C:   6081    ORI    R24,0x01     Logical OR with immediate
+0000003D:   8389    STD    Y+1,R24      Store indirect with displacement

18:      cond.f1=0;
+0000003E:   8189    LDD    R24,Y+1      Load indirect with displacement
+0000003F:   7F8D    ANDI   R24,0xFD     Logical AND with immediate
+00000040:   8389    STD    Y+1,R24      Store indirect with displacement
```

адреса у пам'яті програм — *код команди* — *назва команди* — *аргументи* — *коментар*

б . Порозрядні операції. У алгоритмічній мові Сі для роботи з цілими числами у двійковому представленні є шість порозрядних операцій: & (І), | (АБО), ^ (виключне АБО чи сума за модулем 2), ~ (НЕ),
>> (зсув вправо) та << (зсув вліво). Результат їх виконання для двох бітів наведений у табл. 2 та табл. 3.

Таблиця 2. Таблиця істинності для порозрядних операцій

		Результат			~	
Біт 1	**Біт 2**	**&**	**|**	**^**	**Біт**	**Результат**
0	0	0	0	0	0	1
0	1	0	1	1	1	0
1	0	0	1	1		
1	1	1	1	0		

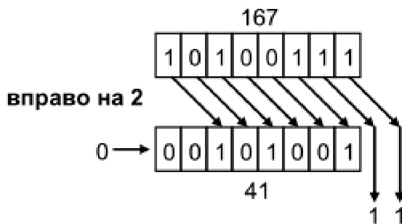
Таблиця 3. Порозрядні операції зсуву

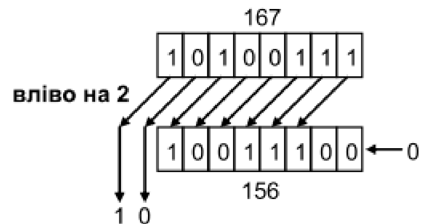
>> зсув вправо	Зсуває біти лівого операнда на число розрядів, що вказане правим операндом. При цьому праві біти втрачаються. Якщо лівий операнд представляє собою ціле число без знаку, то ліві біти, що звільнилися, заповнюються нулями, в іншому випадку, вони заповнюються символом знаку, тобто 1.
<< зсув вліво	Зсуває біти лівого операнда на число розрядів, що вказане правим операндом. При цьому ліві біти втрачаються, а праві заповнюються нулями.

Результати цих операцій для цілих чисел виглядають так:

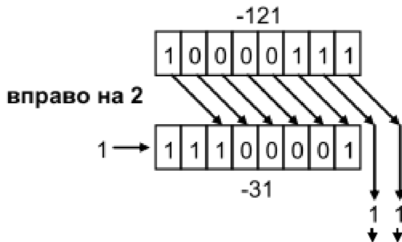
<table>
<tr><td>10</td><td>1 0 1 0</td><td>10</td><td>1 0 1 0</td><td>10</td><td>1 0 1 0</td><td>10</td><td>1 0 1 0</td></tr>
<tr><td>&</td><td>&&&&</td><td>|</td><td>||||</td><td>^</td><td>^^ ^^</td><td>~</td><td>~~~~</td></tr>
<tr><td>9</td><td>1 0 0 1</td><td>9</td><td>1 0 0 1</td><td>9</td><td>1 0 0 1</td><td>↓</td><td>↓↓ ↓↓</td></tr>
<tr><td>↓</td><td>↓↓ ↓↓</td><td>↓</td><td>↓↓ ↓↓</td><td>↓</td><td>↓↓ ↓↓</td><td>5</td><td>01 01</td></tr>
<tr><td>8</td><td>1 0 0 0</td><td>11</td><td>1 0 1 1</td><td>3</td><td>0 0 1 1</td><td></td><td></td></tr>
</table>

Порозрядний зсув беззнакового однобайтного числа 167





Порозрядний зсув знакового однобайтного числа -121



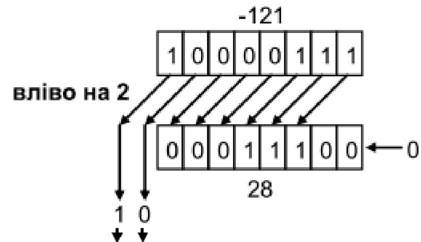


Рис. 1. Ілюстрація роботи порозрядних зсувів

Операція порозрядного зсуву вліво << цілого числа на n розрядів еквівалентна множенню числа на 2^n , при умові що крайні біти не губляться. Наприклад, при зсуві числа 5 (101_2) на 3 розряди вліво отримуємо число 40 (101000_2), що тотожно множенню числа 5 на $8 = 2^3$.

Операція порозрядного зсуву вправо >> цілого числа на n розрядів еквівалентна цілочисельному діленню цього числа на 2^n . Наприклад, при зсуві числа 53 (110101_2) на 3 розряди вправо отримуємо число 6 (110_2), що тотожно цілочисельному діленню числа 53 на $8 = 2^3$ (53 / 8 = 6,625).

Зазначимо, що біти нумеруються справа наліво, причому крайній справа (молодший) біт має номер 0. Довжина біта, півбайта, байта, півслова, слова та подвійного слова рівні, відповідно, 1, 4, 8, 16, 32 та 64 (рис. 2).

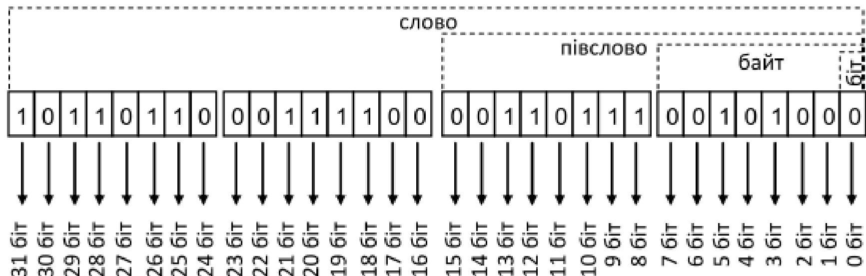


Рис. 2. Внутрішня структура цілого числа та нумерація його бітів

в. Встановлення чи очищення бітів. При роботі з регістрами вводу/виводу (периферії) часто необхідно встановлювати чи скидати окремі біти, чи навіть декілька бітів одночасно. На асемблері для роботи з окремими бітами були відповідні команди sbi та cbi. Рекомендується використовувати для встановлення та скидання окремих бітів такі конструкції мови Сі.

```
unsigned char C=0b10101010;          // глобальна змінна
```

Встановлення окремого біта в «1»:

```
C |= (1<<6);          // встановлення 6-го біта в "1";  C=0b11101010

LDS    R24,0x0060    Load direct from data space
ORI    R24,0x40      Logical OR with immediate
STS    0x0060,R24    Store direct to data space
```

Згенерований дисасемблером код

Скидання окремого біта в «0»:

```
C &= ~(1<<3);         // скидання 3-го біта в "0";   C=0b11100010

LDS    R24,0x0060    Load direct from data space
ORI    R24,0x40      Logical OR with immediate
STS    0x0060,R24    Store direct to data space
```

Згенерований дисасемблером код

Аналогічні маніпуляції і для групи бітів:

```
C |= (1<<4)|(1<<2)|(1<<0);          // встановлення 0,2,4 біта в "1"
C &= ~( (1<<4)|(1<<2)|(1<<0) );     // скидання 0, 2, 4 біта в "0"
```

При роботі з портами вводу/виводу компілятор генерує для встановлення/скидання окремих бітів більш компактний код:

```
DDRA |= (1<<6);                     // встановлення 6-го біта в "1";

SBI     0x1A,6      Set bit in I/O register
```

```
DDRA &= ~(1<<3);                    // скидання 3-го біта в "0";

CBI     0x1A,3      Clear bit in I/O register
```

```
DDRA |= (1<<4)|(1<<2)|(1<<0);       // встановлення 0, 2, 4 біта в "1"

IN      R24,0x1A    In from I/O location
ORI     R24,0x15    Logical OR with immediate
OUT     0x1A,R24    Out to I/O location
```

```
DDRA &= ~((1<<4)|(1<<2)|(1<<0)); // скидання 0, 2, 4 біта в "0"

IN      R24,0x1A    In from I/O location
ANDI    R24,0xEA    Logical AND with immediate
OUT     0x1A,R24    Out to I/O location
```

г. Інверсія бітів. Для інверсії бітів використовується операція виключного АБО (сума за модулем 2).

```
C ^= (1<<2);                        // інверсія 2-го біта змінної C

LDS     R24,0x0060   Load direct from data space
LDI     R18,0x04     Load immediate
EOR     R24,R18      Exclusive OR
STS     0x0060,R24   Store direct to data space
```

```
C ^= (1<<4)|(1<<2)|(1<<0);          // інверсія 0,2,4 біта змінної C
```

Аналогічно і для регістрів вводу/виводу:

```
DDRA ^= (1<<2);                     // інверсія 2-го біта регістра DDRA

IN      R24,0x1A    In from I/O location
LDI     R25,0x04    Load immediate
EOR     R24,R25     Exclusive OR
OUT     0x1A,R24    Out to I/O location
```

```
DDRA ^= (1<<4)|(1<<2)|(1<<0);       // інверсія 0,2,4 біта регістра DDRA
```

д. Перевірка значень бітів. В умовних конструкціях часто необхідно виконувати перевірку значень окремих бітів цілочисельного числа чи регістра вводу/виводу. Для цього ми можемо скористатися або готовим макросом WINAVR, або написати відповідну конструкцію мовою Сі.

```
unsigned char  ivalue = 121;        // глобальна змінна


// якщо 2-й біт у змінній ivalue встановлений, bit2<>0
if( bit_is_set(ivalue, 2) ) ivalue=10;          // використовуючи макрос bit_is_set()

if( ivalue & (1<<2) )       ivalue=10;          // використовуючи Сі-конструкцію

LDS     R24,0x0061     Load direct from data space
SBRS    R24,2          Skip if bit in register set
RJMP    PC+0x0004      Relative jump
LDI     R24,0x0A       Load immediate
STS     0x0061,R24     Store direct to data space

// якщо 2-й біт у змінній ivalue скинутий, bit2=0
if( bit_is_clear(ivalue, 2) )  ivalue=10;       // використовуючи макрос bit_is_clear()

if( !(ivalue & (1<<2)) )       ivalue=10;       // використовуючи Сі-конструкцію

LDS     R24,0x0061     Load direct from data space
SBRC    R24,2          Skip if bit in register cleared
RJMP    PC+0x0004      Relative jump
LDI     R24,0x0A       Load immediate
STS     0x0061,R24     Store direct to data space


// якщо 2-й біт у регістрі DDRA встановлений, bit2<>0
if( bit_is_set(DDRA, 2) ) DDRA=10;              // використовуючи макрос bit_is_set()

if( DDRA & (1<<2) )       DDRA=10;              // використовуючи Сі-конструкцію

SBIS    0x1A,2         Skip if bit in I/O register set
RJMP    PC+0x0003      Relative jump
LDI     R24,0x0A       Load immediate
OUT     0x1A,R24       Out to I/O location

// якщо 2-й біт у регістрі DDRA скинутий, bit2=0
if( bit_is_clear(DDRA, 2) ) DDRA=10;            // використовуючи макрос bit_is_clear()

if( !(DDRA & (1<<2)) )      DDRA=10;            // використовуючи Сі-конструкцію

SBIC    0x1A,2         Skip if bit in I/O register cleared
RJMP    PC+0x0003      Relative jump
LDI     R24,0x0A       Load immediate
OUT     0x1A,R24       Out to I/O location
```

Умова в операторі **if**() справджується, якщо її результат відмінний від нуля, і навпаки, якщо вираз умови повернув значення нуль, тоді умова не справджується. Булеві типи на мові Сі відсутні. Таким чином, ми можемо формувати логічні вирази для перевірки значень одразу декількох бітів. Наприклад:

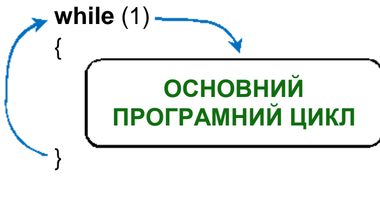
```
// якщо 1й або 3й біти у змінній ivalue встановлені, (bit1==1 ||
bit3==1) if( ivalue & 0b00001010 ) ivalue=10;

// якщо 1й та 3й біти у змінній ivalue встановлені, (bit1==1 &&
bit3==1) if( ivalue & 0b000001010 == 0b00001010) ivalue=10;


// якщо 1й або 3й біти у змінній ivalue скинуті, (bit1==0 || bit3==0)
if( ~ivalue & 0b00001010) ivalue=10;

// якщо 1й та 3й біти у змінній ivalue скинуті, (bit1==0 &&
bit3==0) if( !(ivalue & 0b000001010) ) ivalue=10;
```

Маска 0b00001010 відповідає виразу (1<<3)|(1<<1).

4. Базова структура програми мовою Сі.

```
// перелік підключених бібліотек
#include <avr/io.h>
#include <avr/interrupt.h>

// секція для макровизначень
#define XTAL 8000000L
#define HI(x) ((x) >> 8)        // макрос для отримання ст. байта //
#define LO(x) ((x) & 0xFF)      макрос для отримання мол. байта

// оголошення структур даних та глобальних
змінних struct Time
{
        unsigned char hour;
        unsigned char minute;
        unsigned char second;
};

volatile unsigned char temp;
int value;

// оголошення прототипів користувацьких
функцій void Init(void);
```

```
// підпрограми переривань
ISR(TIMER0_OVF_vect)                    // переривання Таймера Т0 по переповненню
{

}


// основна функція
int main(void)
{
        // оголошення та ініціалізація локальних
        змінних struct Time time1={0, 0, 0};
        int k=0;
        // ініціалізація периферії
        Init();
```




```
}


// користувацькі
функції void Init(void)
{
        DDRA = 0x00;                    // Порт А на вхід
        PORTA = 0xFF;                   // з внутр. підтягуючими резисторами

        DDRB = 0xFF;                    // Порт В на вихід
        PORTB = 0x00;                   // низький рівень
}
```

У налаштуваннях проекту AVR Studio v.4 на основі компілятора мови Сі (WINAVR) обов'язково необхідно вказувати назву МК та частоту тактового генератора, на якій він буде працювати.

У кожному проекті обов'язково слід підключати бібліотеку <avr/io.h>. Завдяки їй компілятор виконає для нашого конкретного МК імпорт усіх назв регістрів вводу/виводу з їхніми адресами, а також назви окремих бітів, що розміщуються у цих регістрах.

Якщо у програмі використовуються переривання, тоді також слід підключити бібліотеку <avr/interrupt.h>.

Макрос ISR() вказує на підпрограму переривання. Як аргумент макросу вказується назва переривання.

Основна функція main (з якої починається робота МК), містить необхідну ініціалізацію периферії та основний робочий безмежний цикл. Як правило, його реалізують у такому вигляді:

```
while (1)
{

}
```

Альтернативним варіантом може бути і такий безмежний цикл

```
for ( ; ; )
{

}
```

Користувацькі функції можемо вказувати як перед основною функцією main, так і після неї. У другому випадку, щоб не було повідомлень компілятора, необхідно перед функцією main оголошувати прототипи наших користувацьких функцій.

Примітка: ініціалізацію стеку компілятор виконує автоматично.

5. Глобальні та локальні змінні, директива volatile.

Змінні, що оголошені у глобальному просторі програми, розміщуються компілятором в оперативній пам'яті SRAM, починаючи з адреси $060. Локальні змінні, оголошені усередині функцій та підпрограм переривань, розміщуються компілятором у стеку.

Підпрограми переривань не приймають ніяких аргументів, тому для передачі їм значень із основної функції та від них в основну функцію, слід використовувати глобальні змінні.

Важливою рисою компілятора Сі є те, що він виконує оптимізацію коду, написаного програмістом, щоб кінцевий згенерований код на асемблері був коротким і швидкодіючим. Однак оптимізація, що виконується компілятором, може спотворити логіку програми, а навіть і зробити її взагалі непрацездатною.

При оптимізації та спрощенню коду компілятор часто викидає куски програмного коду , які на його думку не впливають на роботу МК. Тобто, компілятор може вилучити всі ділянки програми, які опрацьовують змінні, що не змінюють (на думку компілятора) своїх значень.

Наприклад, ми хочемо зробити невеличку програмну затримку:

```
int main(void)
{
	for(int i=0; i<10000; i++) ;
	while (1) { }
}
```

Компілятор розгляне цикл for як такий, що не впливає на роботу МК, і тому він його вилучить на етапі компіляції.

Інший випадок, ми оголошуємо певну глобальну змінну, яка буде використовуватися як в основній програмі, так і в перериванні. В основній програмі ми присвоюємо їй значення нуль, і одразу після цього перевіряємо її у циклі while на рівність нулю. Допоки змінна рівна нулю – цикл має постійно повторюватися, аж до моменту зміни її значення на одиницю, при настанні події переривання таймера T0 по переповненню.

```
unsigned char temp;

ISR(TIMER0_OVF_vect)  // переривання Таймера T0 по переповненню
{
	temp = 1;
}
int main(void)
{// ініціалізація таймера T0
	TCCR0 = (1<<CS02)|(1<<CS00);   // Кпод=1024, по переповненню
	TIMSK |= 1<<TOIE0; // дозвіл на переривання по переповненню
	sei(); // глобальний дозвіл на переривання
	temp = 0;
	while (temp == 0);           // тут програма зациклиться на постійно
	DDRA = 0xF0;                 // компілятор пропустить цей код
	PORTA= 0x0F;                 // компілятор пропустить цей код
	while (1) { }                // компілятор пропустить цей код
}
```

Оскільки змінна temp ініціалізується нулем безпосередньо перед циклом while, а цикл повторюється до тих пір, поки ця змінна рівна нулю, а всередині циклу змінна не змінюється, то компілятор робить відповідний висновок, що тут передбачене постійне зациклення. Все що нижче циклу (ініціалізація порту А і далі) компілятором ігнорується. Хоча в дійсності ми передбачили, що змінна temp змінить своє значення у перериванні таймера, і відбудеться вихід з циклу.

Це пояснюється тим, що компілятор не аналізує код у паралельних процесах (основна програма та підпрограми переривань). Тому такі змінні слід оголошувати з ключовим словом **volatile** (з англ. означає «непостійний»).

Директива volatile у мові Сі вказує компілятору, що значення зазначеної змінної може бути зміненим у будь-який момент, навіть нехай і невідомим для компілятора способом, і що частина коду, яка виконує над цією змінною певні дії (читання чи запис), не повинна бути оптимізованою. Тобто, оголошену таким чином змінну, не можна чіпати при оптимізації.

Для коректної роботи попередніх прикладів необхідно відповідні змінні оголосити з ключовим словом volatile.

```
//======= Для першого прикладу ==========
for(volatile int i=0; i<10000; i++) ;

//======= Для другого прикладу ===========
volatile unsigned char temp;
```

6. Робота з перериваннями.

Підпрограма переривання визначається за допомогою макросу ISR(). Цей макрос, згідно вказаної у дужках назви вектора переривань, реєструє та позначає вказану підпрограму як обробник переривання для визначеного периферійного пристрою. Наступний приклад ілюструє визначення обробника події для переривання від модуля АЦП.

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

ISR(ADC_vect)
{
   // тут вписується користувацький код
}
```

При використанні переривань обов’язково необхідно підключати бібліотеку <avr/interrupt.h>. Ця бібліотека визначає для вказаної моделі МК необхідні адреси векторів переривань.

Якщо переривання для певного периферійного пристрою дозволене, але немає визначеного обробника для цього переривання (слід розглядати це як програмний дефект), тоді при виникненні цієї події буде виконаний скид МК та перенаправлення на вектор RESET.

Для підпрограми обробки події переривання компілятор автоматично генерує код для кешування у стеку регістра стану SREG на

початку підпрограми та його відновлення зі стеку при виході з підпрограми переривання.

У табл. 4 вказаний перелік усіх назв векторів переривань для моделі ATmega32. Ці назви вписуються як аргумент макросу ISR().

Таблиця 4. Перелік назв векторів переривань для ATmega32

Назва вектора	Опис
ADC_vect	ADC Conversion Complete
ANA_COMP_vect	Analog Comparator
EE_RDY_vect	EEPROM Ready
INT0_vect	External Interrupt Request 0
INT1_vect	External Interrupt Request 1
INT2_vect	External Interrupt Request 2
SPI_STC_vect	Serial Transfer Complete
SPM_RDY_vect	Store Program Memory Ready
TIMER0_COMP_vect	Timer/Counter0 Compare Match
TIMER0_OVF_vect	Timer/Counter0 Overflow
TIMER1_CAPT_vect	Timer/Counter1 Capture Event
TIMER1_COMPA_vect	Timer/Counter1 Compare Match A
TIMER1_COMPB_vect	Timer/Counter1 Compare MatchB
TIMER1_OVF_vect	Timer/Counter1 Overflow
TIMER2_COMP_vect	Timer/Counter2 Compare Match
TIMER2_OVF_vect	Timer/Counter2 Overflow
TWI_vect	2-wire Serial Interface
USART_RXC_vect	USART, Rx Complete
USART_TXC_vect	USART, Tx Complete
USART_UDRE_vect	USART Data Register Empty

Для надання дозволу глобального переривання чи його заборони використовуються відповідні макроси sei() та cli(). Також у бібліотеці avr/interrupt.h вказуються й інші макроси для організації переривань.

7. Робота з даними в пам'яті програм.

У бібліотеці <avr/pgmspace.h> визначаються функції та макроси, що дають можливість звертатися до даних, які зберігаються у пам'яті програм, тобто сегменті коду.

Для того, щоб вказати, що певна змінна, масив даних чи об'єкт визначеної структури розміщується в області пам'яті програм, необхідно в описі типу даних використати макрос PROGMEM та проініціалізувати значеннями. При оголошенні даних без цього атрибута, ці дані будуть розміщені в оперативній пам'яті.

```
PROGMEM int A[2][2] = {{1,2},{3,4}};
PROGMEM char Lecturer[] = "Chepiuk Larina";
PROGMEM int num = 0b10101010;
PROGMEM float fvalue = 22.2;
struct Struc
{
        int a;
        char b;
};
PROGMEM struct Struc str_mas[2] = {{22, 55}, {33, 44}};
```

Атрибут PROGMEM може розміщуватися у довільному місці при оголошенні змінної:

```
int PROGMEM A[2][2]={{1,2},{3,4}};
int A[2][2] PROGMEM ={{1,2},{3,4}};
```

Доступ до значень об’єктів, розміщених в області пам’яті програм, виконується лише за допомогою таких макросів

```
pgm_read_byte(адреса)
pgm_read_word(адреса)
pgm_read_dword(адреса)
pgm_read_float(адреса)
```

Наприклад, при звертанні до елементів структури

```
PORTB = pgm_read_word(&str_mas[0].a);
PORTD = pgm_read_byte(&str_mas[0].b);
```

Приклад. Реалізуємо на основі лінійки з 8-ми світлодіодів, підключених до порту С, різні алгоритми їхнього засвічування: 1) 0→1→2→3→4→5→6→7; 2) 7→6→5→4→3→2→1→0; 3) 7&0→ 6&1→5&2 →4&3; 4) 4&3→5&2→6&1→7&0; 5) усі разом блимають. Зміна алгоритму виконується кнопкою, підключеною до виводу PB0.

```
#include <avr/interrupt.h>
#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>

const unsigned char Alg[5][8] PROGMEM = {
{1<<0,  1<<1,  1<<2,  1<<3,  1<<4,  1<<5,  1<<6,  1<<7},        // алг. 1
{1<<7,  1<<6,  1<<5,  1<<4,  1<<3,  1<<2,  1<<1,  1<<0},        // алг. 2
{1<<7|1<<0, 1<<6|1<<1, 1<<5|1<<2, 1<<4|1<<3,
              1<<7|1<<0, 1<<6|1<<1,   1<<5|1<<2, 1<<4|1<<3},  // алг. 3
```

```
{1<<3|1<<4, 1<<2|1<<5, 1<<1|1<<6, 1<<0|1<<7,
                        1<<3|1<<4, 1<<2|1<<5, 1<<1|1<<6, 1<<0|1<<7 },      // алг. 4
{0xFF, 0, 0xFF, 0, 0xFF, 0, 0xFF, 0} };                                      // алг. 5

int main(void)
{
        DDRB = 0x00; PORTB = 0xFF;      // ініціалізація порту B
        DDRC = 0xFF; PORTC = 0x00;      // ініціалізація порту C
        unsigned char N_alg = 0;        // визначає № алгоритму
        while (1)
        {
                for(int i=0; i<8; i++)
                {// якщо кнопка натиснута, то зміна алгоритму if(
                        !(PINB & (1<<0)) ) {N_alg++; _delay_ms(500);}
                        if(N_alg==5) N_alg = 0;
                        // вивід у порт значення з пам’яті програм
                        PORTC = pgm_read_byte( &Alg[N_alg][i] );
                        _delay_ms(500);
                }
        }
}
```

8. Програмні затримки.

Для організації затримок за допомогою програмних циклів використовуються функції бібліотеки <util/delay.h>. Згідно вхідного параметра функції на етапі компіляції обчислюється необхідна кількість пустих програмних циклів, які, згідно заданої тактової частоти, слід виконати, щоб отримати вказану затримку. Тому перед використанням модуля<util/delay.h> необхідно забезпечити такі 2 умови:

- Визначити значення константи F_CPU, що рівне тактовій частоті контролера у герцах.
- Включити оптимізацію при компіляції.

Варто зазначити, що тривалість цих затримок може бути більшою, через можливі переходи на виконання переривань.

Функції модуля:

void _delay_ms(**double** ms) – виконує затримку у ms мілісекунд. Максимально можливе значення (262.14 мсек) / (F _CPU у МГц). Якщо вхідний параметр більший за це значення, тоді відбудеться автоматичне зниження точності витримки інтервалів затримки. Таким чином можна реалізувати затримку до 530 сек (для 8 МГц).

void _delay_us(**double** us) – виконує затримку у us мікросекунд. Максимально можливе значення (768 мксек) / (F_CPU у МГц). Якщо значення затримки виходить більшим, тоді виклик функції автоматично буде перенаправлений у функцію _delay_ms().